ITS
Institut
Teknologi
Sepuluh Nopember

TUGAS AKHIR - TE 141599

SISTEM KOMPUTASI GPU BERBASIS REMOTE VIRTUALIZATION PADA MESIN VIRTUAL

Rizki Bayu Baskoro
NRP 2212100060

Dosen Pembimbing
Dr. Supeno Mardi Susiki Nugroho, ST., MT.
Arief Kurniawan, S.T., M.T.

JURUSAN TEKNIK ELEKTRO
Fakultas Teknologi Industri
Institut Teknologi Sepuluh Nopember
Surabaya 2017

TUGAS AKHIR – TE141599

SISTEM KOMPUTASI GPU BERBASIS REMOTE VIRTUALIZATION PADA MESIN VIRTUAL

Rizki Bayu Baskoro
NRP 2212100060

Dosen Pembimbing
Dr. Supeno Mardi Susiki Nugroho, ST., MT.
Arief Kurniawan, S.T., M.T.

JURUSAN TEKNIK ELEKTRO
Fakultas Teknologi Industri
Institut Teknologi Sepuluh Nopember
Surabaya 2017

FINAL PROJECT – TE141599

GPU COMPUTATIONAL SYSTEM BASED ON REMOTE VIRTUALIZATION ON VIRTUAL MACHINE

Rizki Bayu Baskoro
NRP 2212100060

Advisor
Dr. Supeno Mardi Susiki Nugroho, ST., MT.
Arief Kurniawan, S.T., M.T.

Department of Electrical Engineering
Faculty of Industrial Technology
Sepuluh Nopember Institut of Technology
Surabaya 2017

PERNYATAAN KEASLIAN TUGAS AKHIR

Dengan ini saya menyatakan bahwa isi sebagian maupun keseluruhan Tugas Akhir saya dengan judul "**Sistem Komputasi GPU berbasis Remote Virtualization pada Mesin Virtual"** adalah benar-benar hasil karya intelektual
mandiri, diselesaikan tanpa menggunakan bahan-bahan yang tidak diijinkan dan bukan karya pihak lain yang saya akui sebagai karya sendiri.

Semua referensi yang dikutip maupun dirujuk telah ditulis secara lengkap pada daftar pustaka.

Apabila ternyata pernyataan ini tidak benar, saya bersedia menerima sanksi sesuai peraturan yang berlaku.

Surabaya, Januari 2017

Rizki Bayu Baskoro
NRP. 2210100024

SISTEM KOMPUTASI GPU BERBASIS REMOTE VIRTUALIZATION PADA MESIN VIRTUAL

TUGAS AKHIR

Diajukan guna Memenuhi Sebagian Persyaratan
Untuk Memperoleh Gelar Sarjana Teknik
Pada
Bidang Studi Teknik Komputer dan Telematika
Jurusan Teknik Elektro
Institut Teknologi Sepuluh Nopember

Menyetujui:

Dosen Pembimbing I	Dosen Pembimbing II
Dr. Supeno Mardi S.N., ST., MT.	**Arief Kurniawan, ST., MT.**
NIP: 197003131995121001	**NIP: 197409072002121001**

SURABAYA
Januari, 2017

ABSTRAK

Nama Mahasiswa	:	Rizki Bayu Baskoro
Judul Tugas Akhir	:	Sistem Komputasi GPU berbasis Remote Virtualization pada Mesin Virtual
Pembimbing	:	1. Dr. Supeno Mardi Susiki N., ST., MT. 2. Arief Kurniawan, ST., MT.

Seiring dengan perkembangan dari teknologi pada GPU (*Graphic Processing Unit*), maka konsep GPGPU (*General Purpose Computing using GPU*) semakin banyak digunakan dalam melakukan komputasi yang pada umumnya dilakukan oleh CPU (Central Processing Unit). Dengan kemampuan komputasi tersebut, konsep tersebut dapat diimplementasikan pada jaringan komputasi awan untuk menciptakan sebuah sistem komputasi berperforma tinggi (*high performance computing*). Namun, penerapan tersebut dapat terhalangi oleh terbatasnya sumber daya perangkat keras yang tersedia, serta tidak tersedianya virtualisasi GPU secara langsung pada teknologi mesin virtual yang ada. Tujuan utama dari penelitian ini adalah untuk membangun sistem komputasi virtual berbasis GPGPU dan jaringan komputasi awan dengan menggunakan metode remote virtualization terhadap GPU pada jaringan. Pengujian dilakukan dengan melakukan komputasi perkalian matriks berukuran besar pada CPU dan GPU baik secara langsung maupun virtualisasi, serta melakukan pengukuran *bandwidth* terhadap perpindahan data antara *host memory* dengan *device memory* untuk mendapatkan gambaran performa dari metode virtualisasi jarak jauh (*remote virtualization*). Dari hasil pengujian komputasi yang didapat, penggunaan metode virtualisasi jarak jauh terhadap GPU tampak tidak menunjukkan adanya penurunan performa bila dibandingkan dengan penggunaan GPU secara langsung. Namun demikian, besar *bandwidth* antara memori sistem terhadap memori dari GPU yang divirtualisasikan terbatasi oleh besar *bandwidth* praktikal dari jaringan yang digunakan, sehingga berpotensi mengurangi performa pada model komputasi tertentu yang bergantung pada komunikasi antara CPU dengan GPU.

Kata Kunci : GPGPU (*General Purpose Computing using GPU*), Mesin Virtual, GPU, Performa

Halaman ini sengaja dikosongkan

ABSTRACT

Name : Rizki Bayu Baskoro
Title : *GPU Computational System based on Remote Virtualization on Virtual Machine*
Advisors : 1. Dr. Supeno Mardi Susiki N., ST., MT.
2. Arief Kurniawan, ST., MT.

With the development of GPU (Graphic Processing Unit) technology, the concept of GPGPU (General Purpose Computing using GPU) are increasingly used in computational problems that is generally done by the CPU (Central Processing Unit). With such computing capability, the concept can be implemented on a cloud computing network to create a small high-performance computing system. However, the implementation can be hindered by the limited hardware resources available, and the inavailability of direct GPU virtualization on the existing virtual machine technology. The main objective of this research is to build a virtual computing system based on GPGPU and cloud computing network using remote method on GPU virtualization on the network. Testing is done by computing a large matrix multiplication on the CPU and GPU both directly and virtualized, as well as measuring the bandwidth of data transfer between host memory with device memory to get the performance overview of the remote virtualization method. From the test results obtained, the use of remote virtualization methods on GPU did not show any decrease in performance when compared with the conventional use of the GPU. However, the bandwidth between the system memory to the GPU memory were limited by practical bandwidth of the network architecture used, thus potentially reducing performance on a particular computing model that relies on communication between CPU and GPU.

Keywords : GPGPU (General Purpose Computing using GPU), Virtual Machines, GPU, Performance

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji dan syukur kehadirat Allah SWT atas segala limpahan berkah, rahmat, serta hidayah-Nya, penulis dapat menyelesaikan penelitian ini dengan judul **Sistem Komputasi GPU berbasis Remote Virtualization pada Mesin Virtual**.

Penelitian ini disusun dalam rangka pemenuhan bidang riset di Jurusan Teknik Elektro ITS, Bidang Studi Teknik Komputer dan Telematika, serta digunakan sebagai persyaratan menyelesaikan pendidikan S1. Penelitian ini dapat terselesaikan tidak lepas dari bantuan berbagai pihak. Oleh karena itu, penulis mengucapkan terima kasih kepada:

1. Keluarga, Ibu, Bapak dan Saudara tercinta yang telah memberikan dorongan spiritual dan material dalam penyelesaian buku penelitian ini.
2. Bapak Dr. Tri Arief Sardjono, ST., MT. selaku Ketua Jurusan Teknik Elektro, Fakultas Teknologi Industri, Institut Teknologi Sepuluh Nopember.
3. Secara khusus penulis mengucapkan terima kasih yang sebesar-besarnya kepada Bapak Dr. Supeno Mardi Susiki Nugroho, ST., MT. dan Bapak Arief Kurniawan, ST., MT. atas bimbingan selama mengerjakan penelitian.
4. Bapak-ibu dosen pengajar Bidang Studi Teknik Komputer dan Telematika, atas pengajaran, bimbingan, serta perhatian yang diberikan kepada penulis selama ini.
5. Seluruh teman-teman asisten Laboratorium B201 Telematika dan Elektro angkatan 2012.

Kesempurnaan hanya milik Allah SWT, untuk itu penulis memohon segenap kritik dan saran yang membangun. Semoga penelitian ini dapat memberikan manfaat bagi kita semua. Amin.

Surabaya, Januari 2017

Penulis

Halaman ini sengaja dikosongkan

DAFTAR ISI

TABLE OF CONTENTS

DAFTAR GAMBAR

DAFTAR TABEL

Halaman ini sengaja dikosongkan

DAFTAR KODE

Halaman ini sengaja dikosongkan

BAB 1
PENDAHULUAN

Penelitian ini di latar belakangi oleh berbagai kondisi yang menjadi acuan. Selain itu juga terdapat beberapa permasalahan yang akan dijawab sebagai luaran dari penelitian.

1.1 Latar belakang

Dalam beberapa tahun terakhir, teknologi *high performance computing* (HPC) telah berkembang menjadi sebuah platform heterogen, yang mengintegrasikan penggunaan *multi core processor* (CPU) dengan akselerator komputasi berbasis perangkat keras, seperti GPU. GPU terdiri atas unit komputasi yang lebih kecil, namun berjumlah lebih banyak dari CPU. Selain itu, GPU memiliki memori privat dengan *bandwidth* lebih tinggi. Teknologi terbaru pada GPU telah memberikan kemampuan komputasi *high precision floating-point* serta menggunakan memori yang telah mendukung *error correcting code* (ECC). Sehingga, GPU sering digunakan sebagai akselerator untuk mempercepat proses komputasi pada aplikasi yang memiliki jumlah operasi aritmatik per data serta paralelisasi yang sangat tinggi [1]. Penerapan ini dinamakan General Purpose computing on GPU (GPGPU).

Dengan kemampuan komputasi tersebut, GPGPU dapat diimplementasikan pada jaringan sistem komputasi awan (*cloud computing*) untuk melakukan komputasi berperforma tinggi (*high performance computing*). Hal ini ditunjang oleh kelebihan dari model komputasi awan yang mudah diperluas (*scalable*), tertutup (*private*), mudah digunakan (*ease-of-use*) serta kustomisasi yang luas (*customizability*). Selain itu, teknologi virtualisasi yang semakin berkembang telah meningkatkan performa dari mesin virtual sehingga dapat dibandingkan dengan performa dari *bare-metal system* dalam melakukan berbagai jenis komputasi [2].

Dalam merealisasikan konsep tersebut, tentunya terdapat beberapa batasan yang berpotensi menjadi kendala, sehingga menyebabkan tidak serta-merta dapat direalisasikan. Salah satunya yaitu jumlah GPU yang disediakan untuk tiap *host system*. Apabila se-

buah layanan mesin virtual terdiri atas banyak *host system*, maka jumlah GPU yang harus disediakan sebanyak jumlah dari semua system host tersebut. Selain itu, terdapat berbagai masalah lain diluar beban pengadaan (penambahan GPU), seperti beban listrik, pemeliharaan, dan sebagainya. Perlu diingat pula bahwa seluruh GPU tersebut tidak akan selalu digunakan oleh mesin virtual pada sistem tersebut, sehingga memberikan inefisiensi pada konsumsi daya [3].

Salah satu metode yang dapat dilakukan untuk meminimalisir kekurangan-kekurangan tersebut ialah dengan memvirtualisasikan GPU terhadap mesin virtual pada seluruh jaringan komputasi awan [3]. Dengan metode ini, GPU tidak harus dipasangkan pada seluruh sistem, melainkan pada beberapa saja, sehingga dapat mengurangi kebutuhan perangkat keras serta konsumsi daya keseluruhan secara signifikan. Seluruh mesin virtual dalam jaringan akan dapat mengakses GPU tersebut layaknya perangkat keras milik *host system* miliknya sendiri.

1.2 Permasalahan

Beberapa permasalahan yang menjadi dasar penelitian ini adalah :

1. Kebutuhan akan komputasi berbasis GPU terhalangi oleh terbatasnya perangkat keras GPU yang tersedia. Hal ini akan sangat menyulitkan karena sumber daya perangkat keras (GPU) dalam jumlah banyak belum tentu dapat dipenuhi.

2. Mesin Virtual konvensional tidak memberikan virtualisasi secara langsung pada GPU, sehingga tidak dapat digunakan untuk melakukan komputasi berbasis GPU.

1.3 Tujuan

Tujuan dari dilaksanakannya penelitian ini yaitu :

1. Membangun sistem komputasi awan yang dapat memfasilitasi kebutuhan terhadap sarana komputasional berbasis GPU menggunakan sumber daya perangkat keras (*hardware resource*) yang telah tersedia dengan menggunakan skema *Infrastructure-*

as-a-Service (IaaS) berbasis virtualisasi GPU jarak jauh (*remote GPU virtualization*).

2. Memberikan kemampuan komputasi GPU pada mesin virtual dengan menggunakan metode virtualisasi jarak jauh sebagai salah satu alternatif untuk memberikan kemampuan komputasi tambahan pada mesin virtual.

1.4 Batasan masalah

Adapun batasan masalah yang timbul dari permasalahan Tugas Akhir ini adalah :

1. Jenis komputasi GPU yang akan digunakan pada penelitian merupakan komputasi numerikal dan aritmatika. Komputasi berbasis grafis tidak dapat dilakukan karena limitasi dari GPU yang digunakan.
2. Metode yang digunakan untuk melakukan virtualisasi *GPU* adalah dengan menggunakan rCUDA *middleware framework* yang berbasis Nvidia CUDA.
3. Arsitektur sistem komputasi awan yang digunakan berbasis OpenStack, dengan sistem operasi Ubuntu Server. Penggunaan sistem komputasi awan pada penelitian ini bertujuan untuk mempermudah manajemen serta pembentukan mesin virtual yang digunakan pada pengujian.
4. Arsitektur jaringan yang digunakan pada penelitian berbasis Gigabit Ethernet (1000Mbps).

1.5 Sistematika Penulisan

Laporan penelitian tugas ini disusun secara terstruktur agar isi dari laporan dapat dipahami dan dipelajari oleh pembaca dari berbagai kalangan, serta agar dapat digunakan sebagai acuan dan referensi bagi pembaca yang ingin melanjutkan penelitian ini. Sistematika penulisan laporan penelitian ini terdiri atas :

1. BAB I Pendahuluan
   Bab ini berisi uraian tentang latar belakang dari permasalahan yang dibahas, penegasan dan alasan pemilihan judul, sistematika penulisan laporan, tujuan serta metodologi penelitian.

2. BAB II Tinjauan Pustaka
   Bab ini berisi tentang uraian dari teori-teori serta referensi yang berhubungan dengan permasalahan yang dibahas pada penelitian ini. Teori-teori serta referensi yang didapat tersebut digunakan sebagai acuan dasar dalam melakukan penelitian.
3. BAB III Perancangan Sistem dan Impementasi
   Bab ini berisi penjelasan terkait perancangan serta pembangunan sistem yang akan diuji. Di dalamnya dijelaskan spesifikasi dari perangkat keras, instalasi sistem operasi dan perangkat lunak yang digunakan. Selain itu, juga disertakan alur kerja (*work flow*) serta blok diagram untuk menunjang penjelasan-penjelasan tesebut.
4. BAB IV Pengujian dan Analisa
   Bab ini menjelaskan detil dari pengujian yang dilakukan terhadap sistem yang telah dirancang dan dibangun, serta menganalisa semua hasil dari pengujian yang telah dilakukan, untuk mendapatkan gambaran performa dari sistem.
5. BAB V Penutup
   Bab ini berisi kesimpulan yang diambil dari penelitian dan pengujian yang telah dilakukan. Selain itu, bab ini juga berisi saran serta kritik membangun dari penulis apabila penelitian ini akan dikembangkan lebih lanjut.
6. BAB VI Lampiran
   Bab ini berisi lampiran-lampiran berupa gambar maupun kode program yang bersifat terlalu panjang atau besar untuk diletakkan pada bab-bab sebelumnya.

1.6 Relevansi

Penelitian mengenai konsep virtualisasi pada GPU sudah pernah diujikan oleh tim pengembang dari SnuCL [4] yang menggunakan OpenCL, vCUDA [5] yang berbasis CUDA, serta rCUDA [6] yang digunakan pada penelitian ini. Sementara penelitian tentang penggunaan virtualisasi GPU pada teknologi mesin virtual pernah dilakukan pada sistem virtualisasi berbasis VMware dimana virtualisasi pada GPU dilakukan *host machine* dimana mesin virtual berjalan [7]. Penelitian ini juga menggunakan desain sistem dan jaringan komputasi awan yang telah diujikan pada penelitian sebelumnya [8].

BAB 2
TINJAUAN PUSTAKA

Demi mendukung penelitian ini, dibutuhkan beberapa teori penunjang sebagai bahan acuan dan refrensi. Dengan demikian penelitian ini menjadi lebih terarah.

2.1 *Graphic Processing Unit* (GPU)

Graphic Processing Unit (GPU) merupakan sebuah mikroprosesor yang didesain untuk melakukan proses komputasi yang bertujuan untuk menampilkan citra terhadap sebuah *display output*. GPU bekerja dengan mengolah data citra pada memori untuk kemudian ditampilkan pada sebuah *output* berupa monitor. Sebelum adanya GPU, penampilan citra dilakukan oleh *Central Processing Unit* (CPU). Seiring dengan perkembangan teknologi komputasi grafis yang kompleks seperti permainan elektronik, dan penggunaan antarmuka pengguna berbasis citra gambar (*Graphical User Interface*atau GUI) pada sistem operasi komputer, maka peran tersebut digantikan oleh GPU untuk mengurangi beban kerja dari CPU.

Pada awal perkembangannya, GPU digunakan untuk memproses serta menampilkan citra dua dimensi (2D) saja, seperti antarmuka pengguna dan permainan elektronik sederhana. Seiring dengan perkembangan teknologi permainan elektronik yang pesat, maka GPU mulai mengusung kemampuan akselerasi grafis tiga dimensi (3D), seperti *Transform, Clipping,* dan *Lighting* (TCL). GPU menjalankan program fungsi berupa *shader*.

Pada mikroprosesor dari GPU, terdapat beberapa *hardware shader*, yang berfungsi untuk melakukan manipulasi citra sebelum ditampilkan pada *display output*. Jenis dari *shader* pada GPU terdiri atas :

1. *Pixel Shader*, yang berfungsi untuk melakukan manipulasi terhadap *pixel* dari citra dua dimensi. Beberapa tugas dari *pixel shader* antara lain memberikan gradasi warna, menampilkan efek cahaya, mengolah citra dua dimensi (citra *digital* dan tekstur), serta mengolah video secara terbatas.
2. *Vertex Shader*, merupakan jenis *shader* tiga dimensi (3D) yang

berfungsi untuk melakukan transformasi terhadap data tiga dimensional dari seluruh *vertex* menjadi bidang dua dimensi untuk ditampilkan pada *display output*. *Vertex shader* juga mengolah tingkat kedalaman relatif suatu *vertex* terhadap layar *display*.

3. *Geometry Shader*, merupakan jenis *shader* tiga dimensi yang berfungsi melakukan pengolahan terhadap obyek geometri tiga dimensi. Data vertex dari *vertex shader* dapat diolah menjadi obyek tiga dimensi dengan menambahkan titik maupun garis terhadap vertex, sebelum dirasterasi dan diolah oleh *pixel shader*.
4. *Tessellation Shader* merupakan jenis *shader* yang berfungsi mengolah *polygon mesh* tiga dimensi, yaitu sekumpulan *vertex* yang telah membentuk sebuah obyek 3D. [9]

Dalam menampilkan representasi citra dua dimensi dari suatu obyek tiga dimensi, maka terdapat beberapa proses yang dilakukan pada GPU. Proses tersebut dinamakan *graphic pipeline* atau *rendering pipeline*. Secara berurutan, proses pada *rendering pipeline* terdiri atas :

1. CPU mengirimkan instruksi berupa *shading program* serta data-data geometris dari objek kepada GPU.
2. Pada GPU, data geometris berupa *vertex* ditransformasikan menjadi model objek tiga dimensi.
3. Objek tiga dimensi yang terbentuk dipindahkan menuju koordinat bidang tiga dimensi, kemudian ditransformasikan terhadap posisi kamera (sudut pandang) dari *display output*.
4. Objek kemudian diberikan pencahayaan sesuai dengan tingkat reflektivitas permukaan objek terhadap cahaya serta arah dari sumber cahaya. Proses ini dinamakan *lighting*.
5. Objek pada koordinat kamera tiga dimensi kemudian ditransformasikan menjadi objek bidang dua dimensi. Transformasi dilakukan terhadap letak dan jarak objek dari kamera.
6. Bagian dari objek yang tidak tampak dari sudut pandang kamera akan dibuang dan tidak digunakan pada proses-proses selanjutnya. Proses ini dinamakan *clipping*.
7. Objek selanjutnya diubah menjadi bentuk *raster* atau *pixelated image*.
8. Citra pikselasi hasil rasterasi tersebut kemudian diberi tekstur

sesuai dengan nilai yang telah disimpan pada proses sebelumnya. Tekstur yang digunakan dapat berupa citra *bitmap* dua dimensi, gradasi warna, serta detil spekular (tingkat reflektivitas dan bayangan permukaan dari objek). [10]

Seluruh jenis *shader* tersebut pada awalnya merupakan sebuah *hardware based function*, dimana jumlah dari setiap jenis *shader* dalam mikroprosesor GPU berbeda-beda. Pada umumnya, jumlah *vertex shader* dalam sebuah GPU lebih banyak dibandingkan dengan jumlah *pixel shader*, dengan tujuan agar kemampuan komputasi tiga dimensi pada GPU dapat dimaksimalkan, serta untuk menekan biaya produksi dari GPU itu sendiri, namun memberikan infleksibilitas dalam pembagian beban kerja pada saat GPU hanya melakukan pengolahan 2D, sehingga menyebabkan sejumlah *vertex shader* tidak digunakan.

Untuk mengatasi permasalahan tersebut, maka muncul jenis *shader* baru yang dapat melakukan tugas dari berbagai jenis *shader* tersebut secara universal. Teknologi yang dinamakan *Unified Shader* tersebut bertujuan agar seluruh bagian dari GPU dapat dimaksimalkan untuk berbagai jenis pengolahan citra atau grafis komputer. Sebuah *Unified Shader* bersifat fleksibel dan serbaguna, dapat melakukan berbagai *shader function* yang berbeda melalui *intstruction set* yang telah diberikan pada saat dibutuhkan [11].

2.2 *General Purpose Computing Using GPU*

General-Purpose Computing Using Graphic Processing Unit (*GPGPU*) merupakan istilah yang diberikan terhadap penggunaan *GPU* untuk melakukan komputasi yang umumnya dilakukan oleh *CPU*. Konsep dari *GPGPU* dilatarbelakangi oleh perkembangan teknologi *unified shader* serta kemampuan memproses *floating point* pada *GPU*. Komputasi yang membutuhkan kalkulasi matriks dan vektor lebih mudah dikerjakan oleh *GPU*. Pada konsep GPGPU, GPU *core* bertindak sebagai *parallel processor*, dimana komputasi dilakukan menggunakan satu instruksi, namun dieksekusi oleh banyak *processing core* secara bersamaan.

Pada awal perkembangannya, agar dapat dijalankan menggunakan *GPU*, program terlebih dahulu ditranslasikan ke dalam bentuk grafikal (bentuk *vertex* dan *pixel*) agar dapat dibaca oleh *Graphic Processing API* yang digunakan, seperti *OpenGL* dan *DirectX*

[12]. Seiring perkembangan dan popularitas penggunaanya, mulai bermunculan *library* serta *API* yang mempermudah implementasi dan penggunaan dari konsep *GPGPU*, seperti NVIDIA *CUDA*, Microsoft *DirectCompute*, serta OpenCL [13].

Beberapa contoh aplikasi umum dari konsep *GPGPU* diantaranya :

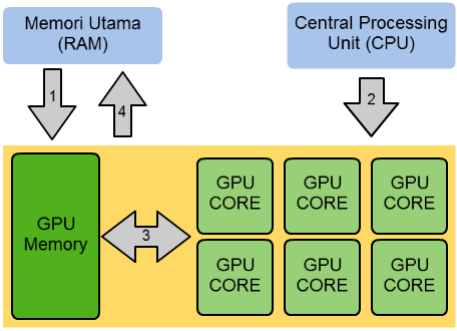
1. Komputasi Saintifik (*Scientific Computing*), dimana *GPGPU* digunakan untuk membantu melakukan simulasi, permodelan, dan algoritma untuk menyelesaikan berbagai macam permasalahan ilmiah menggunakan teknik komputasi yang kompleks, seperti dinamika molekuler (*Molecular Dynamic Simulation*) [14], simulasi propagasi cahaya, dinamika fluida [15], *Deep Machine Learning* [16], simulasi mekanika kuantum, bioinformatika [17], dan lainnya.
2. Pada ilmu kriptografi dan kriptoanalisis, komputasi *GPGPU* dapat digunakan untuk menguji implementasi dari berbagai algoritma enkripsi data, seperti *AES* [18], *MD6*, dan *RSA*. Selain itu, kemampuan komputasi tersebut dapat digunakan untuk melakukan *password-cracking* serta untuk transaksi pemrosesan *cryptocurrency* ("*mining*").
3. *Hardware Accelerated Video Transcoding and Post-processing*, seperti kompresi video, *noise reduction*, teknik *deinterlacing*, koreksi warna, serta *hardware accelerated playback*.
4. dan berbagai aplikasi lainnya.

2.3 CUDA

CUDA merupakan sebuah *Application Programming Interface* yang dikembangkan oleh NVIDIA, yang berfungsi memberikan akses langsung terhadap *GPU* melalui *software layer* yang memiliki serangkaian instruksi tertentu, agar *GPU* dapat digunakan oleh perangkat lunak berbasis *general-purpose computing* [19].

Alur pemrosesan data menggunakan CUDA dapat disesuaikan dengan kebutuhan dari program yang dibuat, namun secara sederhana dapat dijabarkan seperti pada Gambar 2.1 :

1. Data dari memori utama disalin menuji memori *GPU*.
2. *CPU* mengirimkan instruksi pada *GPU* untuk memproses data yang telah disalin pada memori *GPU*.



Gambar 2.1: *Alur pemrosesan data menggunakan CUDA*

3. *GPU* mengeksekusi data secara paralel pada tiap *GPU core*. Data yang telah selesai dieksekusi diletakkan kembali pada memori *GPU*.
4. Data pada memori *GPU* disalin kembali menuju memori utama.

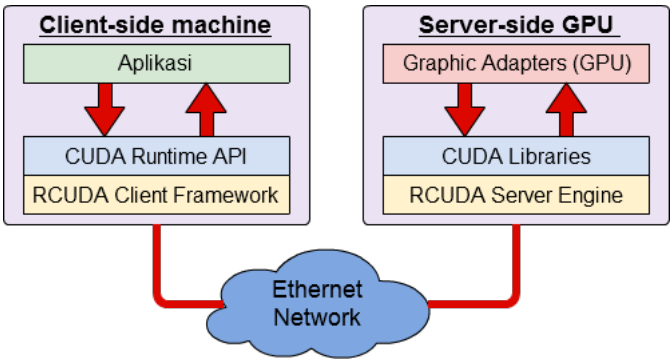
CUDA dapat diimplementasikan melalui CUDA-*accelerated libraries*, *compiler directives*, maupun menggunakan ekstensi dari bahasa pemrograman, seperti C, C++, dan Fortran. Pada bahasa C atau C++, program yang menggunakan CUDA dapat di-*compile* menggunakan *nvcc*. Selain itu, bahasa pemrograman lain seperi Ruby, Python, MATLAB, Perl, Java, dan lain sebagainya juga dapat mengakses fitur dari CUDA melalui *third-party wrapper* yang tersedia bagi masing-masing bahasa.

2.4 RCUDA

RCUDA (Remote-CUDA) merupakan suatu *middleware software framework* yang digunakan untuk melakukan virtualisasi jarak jauh terhadap satu ataupun lebih GPU yang mensupport CUDA melalui jaringan TCP. *GPU* yang terhubung dapat berjalan sebagai sebuah *cluster*, maupun berjalan dalam mesin virtual pada jaringan komputasi awan. Pendekatan ini bertujuan untuk memaksimalkan performa dari *GPU cluster* yang sedang tidak memiliki penggunaan yang penuh. Virtualisasi *GPU* dapat mengurangi jumlah *GPU*

yang dibutuhkan dalam sebuah *computer cluster*, sehingga dapat menekan biaya operasional [3].

RCUDA berjalan pada arsitektur *client-server*. Ketika sebuah *node* yang tidak memiliki *GPU* menjalankan aplikasi yang membutuhkan *GPU*, maka *kernel* akan dieksekusi secara jarak jauh melalui pemindahan data dan code antara memori sistem (node) lokal dengan memori *GPU* yang tertuju. Pada *client*, dijalankan *wrapper library* yang berjalan pada *CUDA Runtime API*, sementara pada *server* terdapat sebuah layanan yang akan melakukan *listening* pada *port* jaringan TCP, menunggu adanya permintaan dari *client*.



Gambar 2.2: *Alur Kerja Virtualisasi Jarak Jauh dengan RCUDA*

Berbagai *node* yang menjalankan aplikasi yang membutuhkan akselerasi *GPU* dapat secara bersamaan menggunakan seluruh *GPU* yang ada pada *cluster* tersebut, dimana tiap *client* mengirim *request* menuju salah satu *server* yang memiliki *GPU*, kemudian dijalankan pada *server* tersebut. Komputasi pada *GPU* tersebut dijalankan menggunakan metode *Time-multiplexing*, dimana *GPU* secara bergantian menjalankan *request* yang diberikan oleh tiap *client node* secara bergantian [6].

2.5 OpenStack

OpenStack merupakan sebuah *software platform* untuk membangun jaringan komputasi awan. OpenStack terdiri atas berbagai komponen yang saling terhubung satu sama lain, yang masing-

masing mengontrol fungsi-fungsi penunjang, seperti penyimpanan data, jaringan, serta pembuatan layanan berbasis IaaS (*Infrastructure as a Service*) [20].

Dalam membangun layanan komputasi awan menggunakan OpenStack, terdapat layanan-layanan inti yang harus dijalankan, masing-masing memiliki fungsi inti yang diperlukan dalam membangun ekosistem komputasi awan, yaitu :

1. OpenStack *Identity Service* (Keystone), berfungsi menyediakan layanan *authentication* dan *authorization* bagi pengguna layanan komputasi awan.
2. OpenStack *Image Service* (Glance), berfungsi menyimpan, mencari, dan mengambil *image* dari mesin virual. Glance juga dapat menggunakan *image* mesin virtual yang telah tersimpan sebagai *template* untuk mesin-mesin virtual berikutnya.
3. OpenStack *Compute* (Nova), sebagai layanan utama pada sistem IaaS (*Infrastructure as a Service*), berfungsi sebagai kontroler dari sistem komputasi awan. Nova didesain untuk memanajemen semua *resource* yang tersedia pada sistem, dan dapat bekerja pada berbagai macam teknologi virtualisasi yang telah ada.
4. OpenStack *Networking* (Neutron), berfungsi mengatur keseluruhan jaringan dari sistem komputasi awan.

Dalam membangun sistem komputasi awan, OpenStack dapat dijalankan dalam berbagai model maupun konfigurasi, sehingga dapat disesuaikan dengan kebutuhan terhadap sistem yang diinginkan. Skema minimal dari OpenStack terdiri atas tiga buah *node*, yang masing-masing menjalankan layanan-layanan inti :

1. *Controller Node*, menjalankan layanan Keystone, Glance, bagian manajemen dari layanan Nova, serta layanan pendukung seperti MySQL *database service* serta RabbitMQ *message broker service*. Pada *Controller Node* juga dapat dijalankan layanan-layanan OpenStack tambahan yang bersifat opsional seperti *Block Storage* (Cinder), *telemetry* (Ceilometer), *Object Storage* (Swift), serta *Orchestration* (Heat).
2. *Compute node* bertindak sebagai tempat dimana mesin virtual akan dijalankan *(Hypervisor)*. *Compute node* menjalankan bagian utama dari Nova. Selain itu, *Compute node* juga men-

jalankan layanan Nova-networking, yang merupakan layanan jaringan OpenStack dengan fitur terbatas yang telah digantikan fungsinya oleh OpenStack Neutron.

3. *Networking node* menjalankan layanan Neutron, yang bertugas mengatur konfigurasi jaringan serta memberikan akses mesin virtual terhadap jaringan luar agar dapat diakses secara publik. *Node* ini tidak digunakan apabila skema jaringan OpenStack menggunakan OpenStack Nova-networking.

BAB 3
DESAIN DAN IMPLEMENTASI SISTEM

Penelitian ini dilaksanakan sesuai dengan desain sistem berikut dengan implementasinya. Desain sistem merupakan konsep dari pembuatan dan perancangan infrastruktur dan kemudian diwujudkan dalam bentuk blok-blok alur yang harus dikerjakan. Pada bagian implementasi merupakan pelaksanaan teknis untuk setiap blok pada desain sistem.

3.1 Desain Sistem

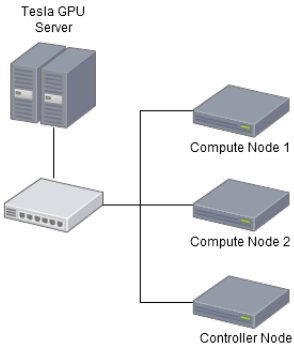
Penelitian ini bertujuan untuk mengukur performa komputasional dari konsep *Remote GPU Virtualization* yang ditawarkan oleh rCUDA terhadap serangkaian mesin virtual pada sebuah jaringan komputasi awan, dan membandingkannya dengan performa *Baremetal System* konvensional. Sebelum pengukuran dilakukan, infrastruktur dari sistem yang akan diuji perlu dibangun terlebih dahulu.

Infrastruktur pertama yang dibangun yaitu sebuah jaringan komputasi awan, yang akan digunakan untuk menyediakan layanan mesin virtual serta fungsi-fungsi administratif yang diperlukan untuk mengatur mesin-mesin virtual tersebut. Dalam membangun infrastruktur tersebut, perangkat lunak yang digunakan adalah OpenStack. Dalam OpenStack, dibutuhkan setidaknya dua buah *server* sebagai kebutuhan minimum untuk menjalankan fungsi layanan mesin virtual. Salah satu *server* berjalan sebagai *controller node*, sementara yang lain beroperasi sebagai *compute node*.

Controller Node berfungsi menjalankan fungsi-fungsi administratif vital yang diperlukan dalam OpenStack, seperti Keystone (*identity service*), Glance (*image service*), cinder (*block storage service*), dan *dashboard*. Selain itu, dalam *controller node* juga terdapat beberapa layanan pendukung seperti MariaDB (*database*) serta *Network Time Server* (NTP). *Controller Node* juga berperan mengatur seluruh *compute node* yang ada dalam sistem. *Compute Node* berfungsi menjalankan *hypervisor*, dimana seluruh mesin virtual

dijalankan. Pada sistem OpenStack, jumlah dari *compute node* dapat disesuaikan dengan jumlah mesin virtual yang dibutuhkan.Baik *Controller Node* maupun *Compute Node* tidak memiliki GPU *(Graphics Processing Unit)*, sehingga mesin-mesin virtual di dalam sistem tersebut tidak dapat melakukan *general-purpose computing* secara langsung.

Infastruktur berikutnya yang akan digunakan ialah *GPU server*. *GPU server* berfungsi menyediakan *remote-GPU* yang nantinya digunakan oleh mesin virtual dalam penelitian ini. *GPU server* terdiri atas dua buah komponen, yakni sebuah *Desktop server* serta sebuah modul Nvidia Tesla S2050 *GPU computing server* yang memiliki *form factor* 1U. Modul tersebut terhubung pada *desktop server* melalui dua buah *Host Interface Card* yang terpasang pada kedua slot *PCI-Express x16* pada *desktop* tersebut.

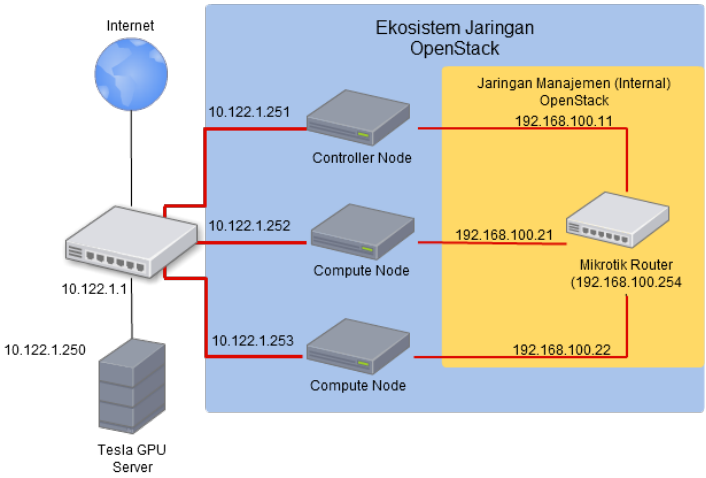


Gambar 3.1: Skema sistem yang digunakan

3.2 Desain Jaringan

Langkah pertama dalam merealisasikan desain sistem adalah perancangan jaringan yang akan digunakan dalam penelitian ini. Pada jaringan komputasi awan berbasis OpenStack, dibutuhkan dua *subnet* yang berbeda. Satu *subnet* digunakan sebagai jaringan manajemen OpenStack, sedangkan jaringan lain bertindak sebagai akses eksternal dari server terhadap jaringan luar. Jaringan manajemen sendiri berfungsi sebagai jalur komunikasi antar layanan yang berjalan di setiap server OpenStack dan tidak dapat digu-

nakan sebagai jaringan eksternal. Pengalokasian IP pada jaringan digambarkan seperti pada gambar 3.2 :

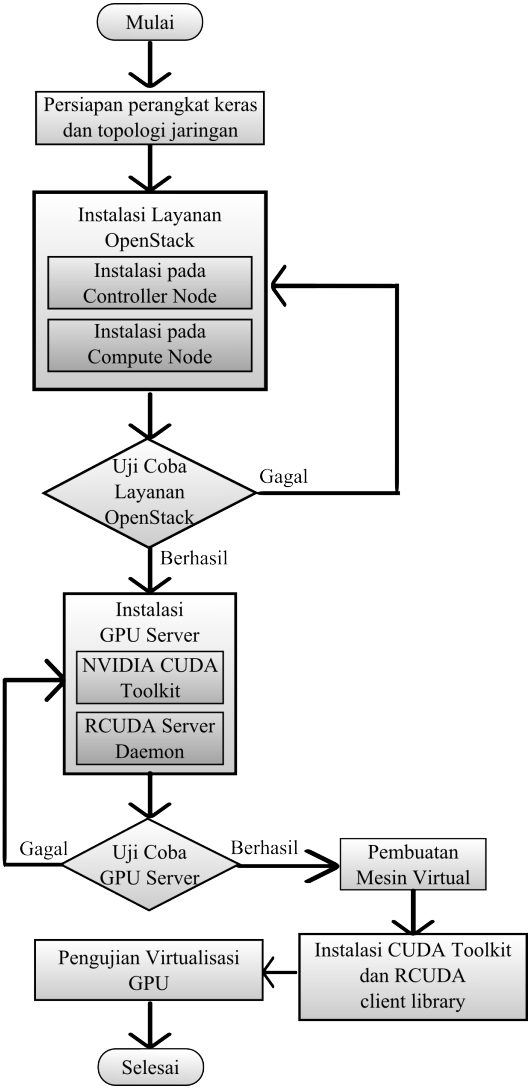


Gambar 3.2: Rancangan sistem yang digunakan

Seperti yang digambarkan pada gambar 3.2, jaringan manajemen untuk OpenStack diletakkan pada subnet 192.168.100.0/24, dimana setiap server saling terhubung melalui sebuah Mikrotik *router*, yang bertindak sebagai *bridge* pada jaringan tersebut. Untuk mempermudah konfigurasi serta manajemen dari Mikrotik *router*, maka *router* tersebut juga diberikan IP jaringan eksternal untuk memberikan akses *remote* terhadap *router* tersebut. Khusus untuk GPU server, tidak diberikan IP pada jaringan manajemen, karena bukan merupakan server penyedia layanan OpenStack.

Pada jaringan eksternal, semua server berada di dalam *subnet* 10.122.1.0/24, dimana tiap server terhubung pada *gateway server*, yang beralamat IP di 10.122.1.1. Jaringan eksternal ini bersifat publik, agar layanan komputasi awan yang berada dalam *subnet* tersebut dapat diakses oleh banyak pengguna, serta agar server-server tersebut dapat diakses secara *remote* oleh administrator dari layanan tersebut.

3.3 Alur Implementasi Sistem

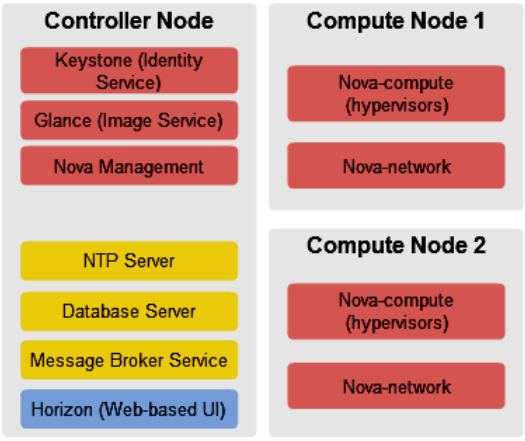


Gambar 3.3: Alur implementasi sistem

Pengerjaan penelitian ini dilakukan dengan melalui delapan tahap, digambarkan pada gambar 3.3. Setiap akhir dari proses instalasi, dilakukan uji coba untuk memastikan seluruh layanan berjalan dengan baik. Apabila layanan tidak berjalan sesuai dengan yang diekspektasikan, maka proses instalasi akan diulang dan diamati satu-persatu untuk mencari penyebab dari kesalahan yang menyebabkan layanan atau fungsi tersebut tidak dapat berjalan.

3.4 Instalasi Layanan Komputasi Awan

Instalasi terhadap layanan komputasi awan dibagi menjadi dua bagian, yakni pemasangan terhadap controller node serta pemasangan dua buah compute node, seperti diilustrasikan pada gambar 3.4.



Gambar 3.4: Skema Instalasi Layanan OpenStack

3.4.1 Persiapan Perangkat Keras dan Sistem Operasi

Pada layanan komputasi awan ini, digunakan tiga buah komputer *server*, dimana satu buah digunakan sebagai *controller node*, dan dua buah lainnya berjalan sebagai *compute node*. Ketiga sistem tersebut menggunakan spesifikasi perangkat keras yang berbeda. Detil dari spesifikasi perangkat keras yang digunakan pada

controller node dijabarkan pada Tabel 3.1.

Tabel 3.1: Spesifikasi dari *controller node*

Spesifikasi	Keterangan
Jumlah Prosesor	2
Jumlah Core	Quad-core
Kecepatan Prosesor	1.86 GHz
Jenis Prosesor	Intel® Xeon® 5120
Arsitektur	x86_64
Memori	2 GB
Disk Storage	160 GB

Pada penelitian ini, terdapat dua buah *compute node*, masing-masing menggunakan spesifikasi perangkat keras yang berbeda. Detil dari spesifikasi perangkat keras yang digunakan pada kedua *compute node* dijabarkan pada Tabel 3.2 dan 3.3.

Tabel 3.2: Spesifikasi dari *compute node 1*

Spesifikasi	Keterangan
Jumlah Prosesor	2
Jumlah Core	Quad-core (4)
Kecepatan Prosesor	1.86 GHz
Jenis Prosesor	Intel® Xeon® 5120
Arsitektur	x86_64
Memori	4 GB
Disk Storage	160 GB

Ketiga *server* tersebut berjalan pada sistem operasi Ubuntu Server 14.04 (Trusty Tahr) dengan arsitektur 64-bit, sedangkan versi dari paket OpenStack yang digunakan adalah OpenStack Juno yang dirilis pada bulan Oktober 2014 untuk menjamin kompatibilitas penuh dengan sistem operasi yang digunakan.

Tabel 3.3: Spesifikasi dari *compute node 2*

Spesifikasi	Keterangan
Jumlah Prosesor	1
Jumlah Core	Hexa-core (6)
Kecepatan Prosesor	1.6 GHz
Jenis Prosesor	Intel® Xeon® E5-2603 v3
Arsitektur	x86_64
Memori	4 GB
Disk Storage	1000 GB

3.4.2 Instalasi *Controller Node*

Sebelum memulai instalasi layanan OpenStack pada *controller node* terlebih dahulu dilakukan instalasi layanan-layanan pendukung terlebih dahulu, seperti layanan *database* dan *message broker*. Kedua layanan tersebut memiliki beragam jenis alternatif yang dapat digunakan, namun pada penelitian ini digunakan perangkat-perangkat lunak berikut :

1. MariaDB sebagai layanan SQL *database*, digunakan untuk menyimpan berbagai *metadata* dari setiap layanan dari OpenStack.
2. RabbitMQ sebagai layanan *message broker* atau *messaging service*, agar setiap layanan OpenStack yang aktif dapat saling berkomunikasi antar *server* yang berbeda.

Setelah layanan-layanan pendukung tersebut terpasang, instalasi layanan utama dari OpenStack dapat dilakukan.

1. *Identity Service* (Keystone)

   Layanan pertama yang akan dijalankan yaitu *identity service*, yang memiliki nama kode **Keystone**. Fungsi dari Keystone terdiri atas *user management* (identifikasi pengguna dan *permission*) serta menyediakan katalog dari layanan yang tersedia menggunakan API (*application programming interface*) dari OpenStack.

   Tiap pengguna serta layanan yang akan ditambahkan pada

OpenStack terlebih dahulu harus didaftarkan pada Keystone. Melalui Keystone, administrator dapat mengatur layanan apa saja yang dapat digunakan oleh pengguna.

2. *Image Service* (Glance)

   Layanan kedua yang dijalankan yaitu *Image Service*, nama kode **Glance**. Glance bertugas melakukan pendaftaran, pencarian, penghapusan serta pengambilan *image* dari mesin virtual. *Metadata* dari *image* yang telah didaftarkan disimpan ke dalam *database* milik Glance, sehingga sebelum instalasi Glance memerlukan sebuah *database* tersendiri untuk menyimpan *metadata* dari layanannya sendiri.

   Seperti layanan lain dari OpenStack, Glance terlebih dahulu harus didaftarkan sebagai layanan pada Keystone. Selanjutnya dilakukan konfigurasi dari Glance untuk menentukan letak dari *database* yang telah dibuat, serta untuk memasukan *authentication* yang telah didapatkan dari Keystone. Semua konfigurasi milik Glance diletakkan dalam sebuah file `/etc/glance/glance-api.conf`. *File* tersebut berisi konfigurasi *messaging service* yang digunakan, letak dan autentikasi dari *database* serta autentikasi layanan terhadap Keystone. Isi dari *file* tesebut dijabarkan pada Kode 3.1.

```
[DEFAULT]
...
# Tampilkan semua output yang terjadi pada logfile
verbose = True

[database]
...
# Konfigurasi lokasi dan autentikasi akses database
connection = mysql://glance:glancedb_pass@controller/
    glance

[keystone_authtoken]
# Konfigurasi autentikasi layanan Glance pada Keystone
auth_uri = http://controller:5000/v2.0
identity_uri = http://controller:35357
admin_tenant_name = service
admin_user = glance
admin_password = glance_admin
```

```
revocation_cache_time = 10
```

Kode 3.1: Isi *file* Konfigurasi OpenStack Glance

3. *Compute Service* (Nova)

   Compute Service atau **Nova** merupakan bagian utama dari OpenStack, menyediakan layanan IaaS (*Infrastructure-as-a-Service*) pada jaringan komputasi awan. Nova bertugas mengatur semua *hardware pool* yang tersedia pada seluruh *compute node* untuk digunakan sebagai *hypervisor* dari mesin virtual.

   Pada *controller node*, komponen dari Nova yang dijalankan bertugas mengatur seluruh komponen Nova yang berjalan pada setiap *compute node* yang terdapat pada keseluruhan jaringan sistem. Beberapa komponen Nova yang berjalan pada *controller node* antara lain :

   (a) Nova-api, berfungsi menerima dan merespon perintah dan panggilan dari pengguna layanan, seperti permintaan untuk membangun mesin virtual, melihat daftar *image* yang tersedia (berkoordinasi dengan Glance-api), dan perintah administratif lain yang berkaitan dengan Nova.

   (b) Nova-api-metadata, berfungsi menerima permintaan *metadata* dari mesin virtual, seperti *image*, dan konfigurasi jaringan.

   (c) Nova-scheduler, bertugas menerima perintah mesin virtual dari daftar antrian (*queue*), serta menentukan pada *compute node* mana mesin virtual akan dijalankan.

   (d) Nova-conductor, bertugas menghubungkan layanan nova-compute dengan *database*.

   (e) Nova-consoleauth, bertugas memberikan autentikasi terhadap pengguna fitur *console proxies*, agar pengguna dapat mengakses konsol milik mesin virtual secara jarak jauh (*remote*).

   Seperti pada saat pemasangan layanan Glance, terlebih dahulu harus dibuatkan *database* tersendiri, dan didaftarkan pada

Keystone. Seluruh konfigurasi layanan Nova pada *controller node* terletak pada `/etc/nova/nova.conf`. *File* tersebut berisi konfigurasi *messaging service* yang digunakan, letak dan autentikasi dari *database*, letak dari layanan Glance pada jaringan sistem, alamat IP yang digunakan sebagai jaringan manajemen, serta autentikasi layanan terhadap Keystone. Penjabaran dari isi *file* konfigurasi tersebut dijabarkan pada Kode 3.2.

```
[DEFAULT]
...
# Tampilkan semua output yang terjadi pada logfile
verbose = True
...
# Pengaturan koneksi pada RabbitMQ (Messaging Service)
rpc_backend = rabbit
rabbit_host = controller
rabbit_password = rabbitmq_pass
...
# Pengaturan alamat IP jaringan manajemen
my_ip = 192.168.100.11
vncserver_listen = 192.168.100.11
vncserver_proxyclient_address = 192.168.100.11
...
# Pengaturan layanan jaringan pada OpenStack
network_api_class = nova.network.api.API
security_network_api = nova

[database]
...
# Pengaturan lokasi dan autentikasi akses database
connection = mysql://nova:novadb_pass@controller/nova

[keystone_authtoken]
# Pengaturan autentikasi layanan Nova pada Keystone
[keystone_authtoken]
auth_uri = http://controller:5000/v2.0
identity_uri = http://controller:35357
admin_tenant_name = service
admin_user = nova
admin_password = nova_admin

[glance]
# Pengaturan letak layanan Glance
host = controller
```

Kode 3.2: Konfigurasi OpenStack Nova pada Controller Node

4. OpenStack Dashboard (Horizon)

   OpenStack Dashboard (nama kode **Horizon**) merupakan layanan penyedia antarmuka berbasis halaman *web*, yang berfungsi untuk memudahkan administrator dan pengguna dalam mengakses layanan dari OpenStack. Melalui antarmuka ini, administrator dapat meninjau dan mengatur seluruh sumber daya dan layanan yang tersedia dalam keseluruhan jaringan sistem, serta melakukan manajemen pengguna seperti pembuatan akun baru dan pengaturan kuota pengguna. Pengguna juga dapat melakukan pembuatan mesin virtual baru atau konfigurasi terhadap mesin virtual yang sedang berjalan. Halhal tersebut dapat dilakukan karena Horizon langsung terhubung dan berinteraksi dengan API dari layanan-layanan dari OpenStack yang telah berjalan.

   Horizon membutuhkan layanan pendukung lain agar dapat berjalan, seperti Apache2 (web server), Memcache, serta Python yang mendukung Django. Horizon sebaiknya dipasang setelah semua layanan OpenStack yang dibutuhkan pada setiap *node* telah berjalan. Setelah Horizon terpasang, dilakukan konfigurasi untuk mengubah beberapa parameter agar Horizon dapat diakses secara leluasa oleh pengguna melalui jaringan.

3.4.3 Instalasi *Compute Node*

Pada kedua *compute node*, layanan dari OpenStack dapat langsung dipasang, karena setelah dilakukan konfigurasi, layanan tersebut akan secara otomatis berkomunikasi dengan layanan yang terdapat pada *controller node*, seperti Keystone, Glance, *database server* serta *messaging server*.

Dalam *compute node*, layanan dari OpenStack yang dipasang hanyalah *Compute Service* atau **Nova**. Komponen dari Nova yang harus dijalankan pada *compute node* yakni nova-compute, sementara komponen lain seperti nova-network dijalankan apabila model

jaringan menggunakan OpenStack Networking Service (Neuron) tidak digunakan.

Konfigurasi dari Nova-compute milik *compute node* disimpan pada file `/etc/nova/nova.conf`. Di dalam *folder* tersebut, juga terdapat *file* `nova-compute.conf` yang berfungsi menyimpan pengaturan dari *hypervisor* yang digunakan pada *compute node* tersebut. Isi dari *nova.conf* serta *nova.conf* pada *compute node* dijabarkan pada Kode 3.3 dan 3.4.

1. *Nova-compute*

   Nova-compute bertugas menjalankan pembuatan dan penghapusan mesin virtual. *Hypervisor* API yang dapat digunakan oleh nova-compute bermacam-macam, seperti XenAPI untuk *hypervisor* berbasis Xen, libvirt untuk QEMU dan KVM, VMwareAPI untuk VMware, dan sebagainya. Nova-compute harus dipasang di setiap *compute node* yang digunakan.

```
[DEFAULT]
...
# Tampilkan semua output yang terjadi pada logfile
verbose = True
...
# Pengaturan koneksi pada RabbitMQ (Messaging Service)
rpc_backend = rabbit
rabbit_host = controller
rabbit_password = rabbitmq_pass
...
# Pengaturan alamat IP jaringan manajemen
my_ip = 192.168.100.22
...
# Pengaturan VNC server untuk akses mesin virtual
vnc_enabled = True
vncserver_listen = 0.0.0.0
vncserver_proxyclient_address = 192.168.100.22
novncproxy_base_url = http://controller:6080/vnc_auto.
    html
...
# Pengaturan layanan nova-network
network_api_class = nova.network.api.API
security_group_api = nova
firewall_driver = nova.virt.libvirt.firewall.
    IptablesFirewallDriver
...
```

```
# Pengaturan metode pembagian alamat IP mesin virtual
network_manager = nova.network.manager.FlatDHCPManager
network_size = 254
allow_same_net_traffic = False
multi_host = True
send_arp_for_ha = True
share_dhcp_address = True
force_dhcp_release = True
flat_network_bridge = br100
flat_interface = eth0
public_interface = eth0

[keystone_authtoken]
# Pengaturan autentikasi layanan Nova pada Keystone
[keystone_authtoken]
auth_strategy = keystone
auth_uri = http://controller:5000/v2.0
identity_uri = http://controller:35357
admin_tenant_name = service
admin_user = nova
admin_password = nova_admin

[glance]
# Pengaturan letak layanan Glance
host = controller
```

Kode 3.3: Konfigurasi OpenStack Nova pada Compute Node

2. *Nova-network*

   Nova-network merupakan komponen dari Nova, bertugas memberikan *network bridge* untuk menghubungkan *physical interface* milik *compute node* dengan mesin virtual agar dapat diakses dari jaringan publik. Setiap kali mesin virtual baru dibuat, Nova-network akan memberikan alamat IP dari *pool* yang telah ditentukan terhadap *instance* yang telah terbentuk. Alamat IP tersebut akan terus digunakan hingga mesin virtual diterminasi oleh pengguna. Dengan menggunakan Nova-network, *compute node* akan bertindak sebagai sebuah *router* yang meneruskan koneksi dari dan menuju mesin virtual dengan jaringan publik luar.

```
[DEFAULT]
```

```
# Pengaturan driver Hypervisor
compute_driver=libvirt.LibvirtDriver

[libvirt]
# Pengaturan jenis Hypervisor yang digunakan
virt_type=kvm
```

Kode 3.4: Konfigurasi Hypervisor Nova-compute

3.5 Instalasi *GPU Server*

Setelah layanan komputasi awan telah berjalan, langkah yang dilakukan selanjutnya yaitu mempersiapkan GPU *server*. GPU *server* terdiri atas dua buah bagian, yakni sebuah *host server* serta sebuah GPU *computing server*. GPU *server* tersebut terhubung dengan layanan komputasi awan melalui jaringan IP publik, agar mesin-mesin virtual dapat saling berkomunikasi secara langsung dengan GPU *server*.

3.5.1 Persiapan Perangkat Keras dan Sistem Operasi

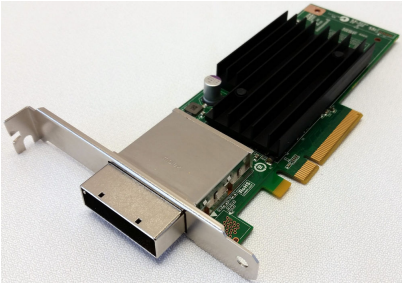
Host server yang digunakan memiliki spesifikasi yang dijabarkan pada Tabel 3.4. Kecepatan dari prosesor diturunkan, serta fitur *hyper-threading* telah dinonaktifkan selama pengukuran performa, agar hasil yang didapatkan dapat diperbandingkan secara langsung dengan performa mesin virtual. Seperti pada *server* layanan komputasi awan, sistem operasi yang digunakan menggunakan Ubuntu 14.04 (Trusty Tahr) dengan arsitektur 64-bit.

GPU *Computing Module* yang digunakan yaitu NVIDIA Tesla S2050. GPU tersebut terhubung melalui dua buah *Host Interface Card* (Gambar 3.5) yang terpasang pada kedua slot PCI-Express x8 pada *host server*, serta bersifat *headless* karena tidak memiliki *video output*. Oleh karena itu, dalam mengakses *desktop server* secara lokal, diperlukan GPU tambahan lain sebagai *video output*.

NVIDIA Tesla S2050 GPU *Computing Module* terdiri atas 4 buah GPU yang terpasang di dalam sebuah *form factor* 1U, sehingga dapat dipasang pada rak server. Modul terhubung pada *host server* melalui dua buah kabel konektor *proprietary* yang tersambung dengan kedua *Host Interface Card*, agar modul dapat dibaca oleh

Tabel 3.4: Spesifikasi dari *GPU Host Server*

Spesifikasi	Keterangan
Jumlah Prosesor	1
Jumlah Core	Quad-core
Kecepatan Prosesor	1.86 GHz
Jenis Prosesor	Intel® Core™ i7 920
Arsitektur	x86_64
Memori	11 GB
Disk Storage	1000 GB



Gambar 3.5: NVIDIA Tesla Host Interface Card

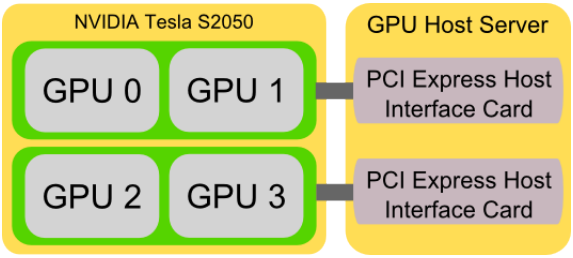
server (diilustrasikan pada gambar 3.6). GPU *core* dari Tesla S2050 berbasis arsitektur Fermi, dan didesain secara penuh untuk keperluan komputasional. Spesifikasi teknis dari NVIDIA Tesla S2050 dapat dilihat pada Tabel 3.5.

3.5.2 Instalasi CUDA *Driver* dan *Library*

Setelah perangkat keras dan sistem operasi telah berjalan dengan baik, maka langkah yang perlu dilakukan selanjutnya adalah memasang driver dari NVIDIA Tesla, serta *library* CUDA pada *host server*, agar CUDA dapat mendeteksi dan berjalan pada GPU *Computing Module* yang digunakan. CUDA *toolkit* dapat diunduh secara langsung dari halaman web milik NVIDIA, namun sebelumnya harus dipastikan terlebih dahulu, apakah versi CUDA *compute* dari GPU telah didukung oleh versi CUDA *toolkit* yang akan digunak-

Tabel 3.5: Spesifikasi NVIDIA Tesla S2050

Spesifikasi	Keterangan
Mikroarsitektur	NVIDIA Fermi
Jumlah GPU	4
Jumlah Memori	12 GB (4 x 3 GB) GDDR5
Jumlah CUDA Core	1792 (4 x 448)
Kecepatan Core	1.15 GHz
Kecepatan Memori	1.55 GHz
Bandwidth Memori	148 GB/s
Interface Memori	384-bit
Versi CUDA Compute	2.0
Max. Single-Precision Floating Point Performance	4.12 GFLOPS



Gambar 3.6: Ilustrasi koneksi Modul NVIDIA Tesla S2050 terhadap *host server*.

an. CUDA *toolkit* versi terbaru memiliki fungsi-fungsi baru yang belum tentu dapat berjalan pada perangkat keras lama, sementara versi CUDA *toolkit* yang terlalu lama hanya dapat mendukung GPU dengan mikroarsitektur dari generasi sebelumnya. Keduanya dapat menyebabkan masalah kompatibilitas serta penurunan performa pada berbagai skenario komputasi dengan perangkat keras yang digunakan.

Terdapat 4 bentuk paket instalasi yang digunakan. Tiga diantaranya bersifat spesifik terhadap distribusi sistem operasi yang

digunakan, yakni *file* `.deb` untuk Linux Debian serta derivasinya (Ubuntu), *file* `.rpm` untuk Linux Red Hat serta derivasinya (Fedora, OpenSUSE), serta *file* `.exe` yang digunakan pada Windows. Paket instalasi terakhir (*file* `.run`) dapat dijalankan pada berbagai distribusi Linux, tanpa menggunakan *package manager* apapun.

Pada paket instalasi CUDA *toolkit* berbasis *file* `.run`, telah disertakan juga *driver package* dari GPU yang telah mendukung versi dari CUDA *toolkit* yang digunakan, sehingga mempersingkat proses instalasi secara keseluruhan.

Pada penelitian ini, CUDA *toolkit* yang digunakan yaitu versi 6.0 serta 6.5. CUDA *toolkit* 6.0 menyertakan driver versi 331.20, yang merupakan versi terakhir yang secara resmi didukung oleh NVIDIA Tesla S2050, sementara CUDA 6.5 merupakan versi pertama yang secara resmi didukung oleh NVIDIA untuk berjalan pada sistem operasi Ubuntu 14.04, serta menyertakan versi driver yang lebih baru (340.93). Kedua *library* tersebut dapat dipasangkan secara berdampingan pada *folder* yang berbeda, sehingga pada saat proses kompilasi kode CUDA, *compiler* dapat diarahkan pada versi *library* yang ingin digunakan. CUDA 6.0 diletakkan pada `/usr/local/cuda-6.0`, sementara CUDA 6.5 diletakkan pada `/usr/local/cuda-6.5`. Saat instalasi, versi driver yang digunakan harus menggunakan versi bawaan CUDA 6.5.

Setelah instalasi, direktori *library* serta *executable* CUDA harus ditambahkan terlebih dahulu pada variabel sistem `PATH` serta `LD_ LIBRARY_PATH` agar dapat terbaca oleh sistem.

3.6 Instalasi rCUDA *Framework*

Pada penelitian ini, versi dari rCUDA yang digunakan merupakan versi 15.07, yang telah mendukung CUDA versi 6.0, 6.5, serta 7.0. Instalasi rCUDA dibagi menjadi dua bagian, yakni pada bagian GPU *server* serta *client (remote) server*. Pada GPU *server*, setelah CUDA *toolkit* telah terpasang, maka langkah yang perlu dilakukan selanjutnya cukup menjalankan *file* `rCUDAd`, yakni *server daemon* dari rCUDA *framework*. Secara *default*, rCUDAd berjalan sebagai *background application*, namun juga dapat dijalankan dalam mode interaktif atau *verbose* dimana informasi yang ditampilkan oleh program tersebut dapat diperhatikan oleh pengguna.

Pada bagian *remote client*, rCUDA terdiri atas beberapa *sha-*

red library yang digunakan untuk menggantikan *library* milik CUDA *toolkit*. Pada saat program dikompilasi, *compiler* yang digunakan (baik `nvcc` maupun *gcc/g++*) harus diarahkan untuk menggunakan rCUDA *shared library* tersebut. Selain itu, perlu ditentukan beberapa *environment variables* berisi alamat IP dari seluruh rCUDA *server* yang terdapat pada jaringan, serta jumlah dari perangkat CUDA yang tersedia pada seluruh GPU *server*, yang dapat digunakan oleh *remote client*.

BAB 4
PENGUJIAN DAN ANALISA

Pada bab ini pengujian dibagi menjadi empat tahapan yaitu pengujian layanan komputasi awan (OpenStack *service*), pengujian GPU *server*, pengujian pembuatan mesin virtual, serta pengujian kinerja mesin virtual dengan virtualisasi GPU tersebut dibandingkan dengan kinerja komputasi dengan CPU. Dengan adanya pengujian-pengujian tersebut, maka akan dapat ditarik beberapa kesimpulan dari pembuatan sistem tersebut.

4.1 Pengujian Layanan Komputasi Awan

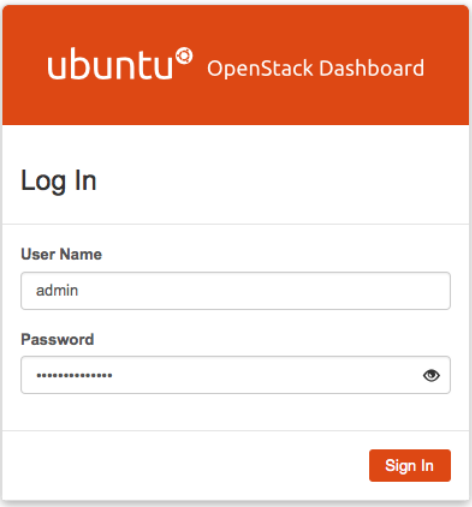
Pengujian ini dilakukan untuk memastikan semua layanan komputasi awan telah berjalan dengan baik sesuai dengan yang diharapkan. Pengujian ini dibagi menjadi beberapa bagian, masing-masing menguji setiap layanan inti yang ada pada jaringan OpenStack, yakni Keystone, Glance, serta Nova.

Pengujian fungsi-fungsi dari layanan tersebut dilakukan melalui OpenStack Horizon, yakni antarmuka berbasis web yang memberikan pengguna serta administrator akses terhadap beberapa perintah serta fungsi utama dari semua layanan tersebut melalui *web browser*. Namun begitu, beberapa perintah tidak tersedia pada antarmuka Horizon, sehingga untuk menjalankan perintah tersebut harus dilakukan secara *remote* secara langsung pada *node server* dimana layanan tersebut berjalan.

4.1.1 OpenStack Keystone (*Identity Service*)

Pengujian layanan Keystone dimulai dengan melakukan *login* melalui antarmuka Horizon menggunakan *username* serta *password* administrator yang telah ditentukan pada proses instalasi layanan. Pada saat pengguna atau administrator mengakses halaman web Horizon, maka tampilan pertama yang akan muncul adalah halaman *Login* seperti pada gambar 4.1.

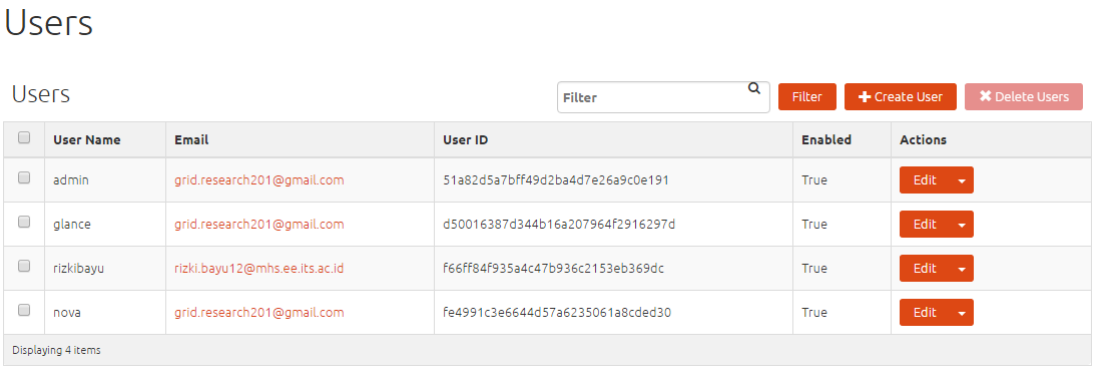
Dengan memasukan *username* serta *password* yang telah ditentukan sebelumnya, maka administrator dibawa menuju halaman **Overview**, dimana ringkasan dari penggunaan keseluruhan layanan komputasi awan terlihat. Melalui menu Admin di bagian kiri



Gambar 4.1: Tampilan halaman *Login* OpenStack Horizon

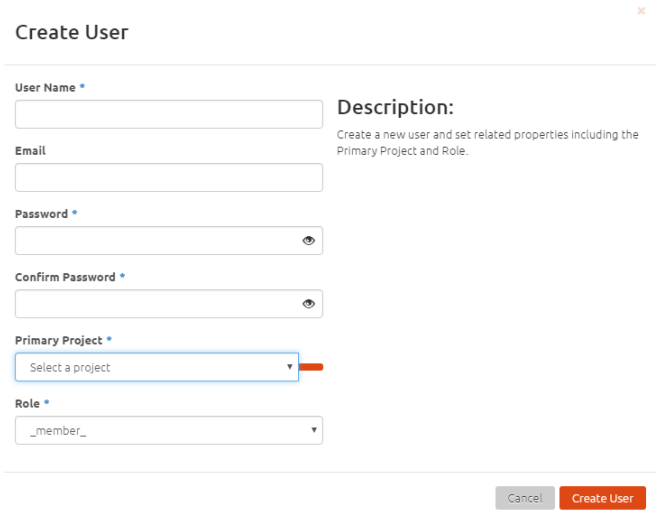
halaman inilah administrator dapat melakukan beberapa perintah administratif terhadap berbagai layanan yang ada pada OpenStack. Menu tersebut tidak akan muncul apabila akun yang digunakan merupakan akun pengguna biasa.

Selanjutnya pengujian dilanjutkan dengan melakukan pemeriksaan terhadap seluruh akun pengguna serta akun layanan yang ada, dengan memilih menu **Identity**, dilanjutkan dengan memilih menu **Users** seperti pada gambar 4.2. Pada halaman tersebut, administrator dapat membuat akun pengguna baru, mengubah, menghapus atau menonaktifkan akun yang telah tidak dipakai. Dapat diperhatikan pula pada gambar 4.2, terdapat beberapa akun milik layanan lain pada OpenStack, yang digunakan oleh layanan-layanan tersebut untuk melakukan autentikasi terhadap Keystone.



Gambar 4.2: Tampilan daftar pengguna

Untuk melakukan pembuatan akun pengguna baru, pilih tombol **Create User**, maka akan muncul formulir pembuatan akun baru seperti pada gambar 4.3.



Gambar 4.3: Tampilan formulir pembuatan akun

Pada formulir tersebut, administrator mengisikan data akun pengguna baru (*username, email*, serta *password*), jenis *tenant* atau *project*, serta *role* dari akun tersebut (akun biasa atau akun administrator). Untuk kepentingan pengujian, maka akan dibuat akun baru dengan *username* serta *password* berisi kata "Tester", jenis akun "member". Setelah selesai dibuat, maka akun tersebut akan muncul pada daftar **Users** pada halaman Horizon.

Pengujian Keystone selanjutnya adalah membuat *tenant* atau *project*. *Tenant* merupakan sebuah pengelompokan terhadap pengguna sebagai bentuk pembatasan sumber daya yang dapat digunakan oleh pengguna layanan OpenStack. Tujuan dari pengelompokan tersebut ialah untuk membatasi besar sumber daya komputasi awan yang dapat digunakan oleh sekelompok pengguna. Contohnya ialah pembuatan *tenant* khusus untuk layanan dari OpenStack yang terpisah dari *tenant* milik pengguna, agar sumber daya yang digu-

nakan kedua *tenant* tersebut tidak tercampur aduk. Setiap pengguna yang terdaftar dalam *tenant* tersebut akan saling berbagi sumber daya yang telah ditentukan sebelumnya oleh Administrator pada saat pembuatan *tenant*. Pembuatan *tenant* atau *project* baru hanya dapat dilakukan oleh akun administrator.

Pembuatan *project* dilakukan melalui menu **Identity - Project**, maka akan tampil daftar *project* yang dapat dialokasikan terhadap pengguna. Pada halaman tersebut, administrator dapat melihat dan mengubah alokasi *resources* serta pengguna yang terdaftar pada *project* tersebut. Selain itu, administrator juga dapat menonaktifkan (*disable*) atau menghapus *project* yang telah tidak digunakan.

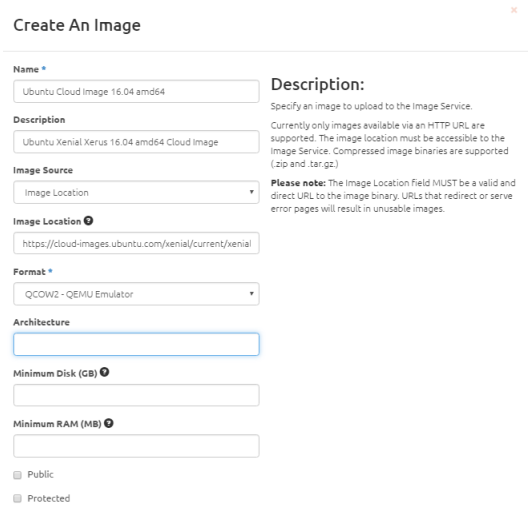
4.1.2 Glance (*Image Service*)

Glance merupakan layanan pada OpenStack, yang bertugas menyediakan *image* dari sistem operasi yang akan digunakan pada mesin virtual. Pada layanan ini, pengujian yang akan dilakukan adalah penambahan serta penghapusan *image* dari *database* milik Glance.

Pada Horizon, pengaturan administratif terhadap Glance terletak pada menu **Admin - System - Images**. Pada halaman tersebut, akan terlihat daftar seluruh *image* sistem operasi serta *snapshot* dari mesin virtual yang tersedia pada Glance.

Penambahan *image* melalui Horizon dapat dilakukan dengan memilih tombol **Create Image**, yang akan menampilkan formulir penambahan *image* seperti pada gambar 4.4. Pada formulir tersebut, administrator dapat menambah *image* menggunakan dua metode, yakni dengan mengunggah *image file* secara manual maupun melalui *direct* HTTP URL halaman web tempat *image file* berasal. Alamat HTTP URL yang digunakan untuk mengunduh *image file* sistem operasi *cloud* dapat dilihat dari halaman web resmi milik sistem operasi yang diinginkan, seperti Ubuntu (`https://cloud-images. ubuntu.com/`).

Image yang akan ditambah juga dapat diatur agar bersifat publik (dapat diakses seluruh pengguna) dan *protected* (hanya dapat dihapus oleh administrator). *Image* yang berhasil diunggah akan muncul pada daftar *Image* seperti pada gambar **??**.



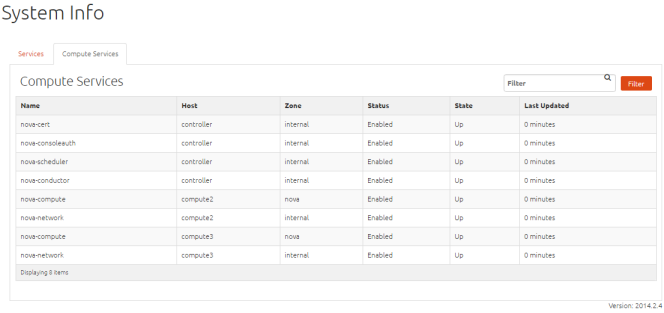
Gambar 4.4: Tampilan Formulir penambahan *Image*

4.1.3 Nova (*Compute Service*)

Pengujian pada layanan OpenStack *Compute Service* (Nova) terbagi menjadi dua bagian utama, yakni pengecekan terhadap seluruh layanan *Compute Service* pada OpenStack, serta pembuatan mesin virtual baru pada kedua *Compute Node*, yang akan dijelaskan pada subbab berikutnya. Tujuan dari kedua pengujian tersebut adalah untuk memastikan seluruh fungsi dari Nova telah berjalan dengan baik.

Pengujian pertama yang dilakukan yaitu memeriksa semua bagian Nova pada seluruh *node* telah berjalan. Dengan menggunakan akun administrator, pengecekan dapat dilakukan melalui Horizon dengan memilih menu **Admin - System** lalu pilih **System Information**. Pada tab **Compute Services**, akan tampil seluruh layanan milik Nova yang telah berjalan, serta *host* letak layanan tersebut berjalan seperti yang diperlihatkan pada gambar 4.5. Pada pengecekan ini, indikator yang diperhatikan yaitu seluruh layanan *service* Nova yang bersifat manajerial berjalan pada *controller node*, sedangkan layanan *hypervisor* dan jaringan mesin virtual (nova-

compute dan nova-network) harus berjalan pada seluruh *compute node*.

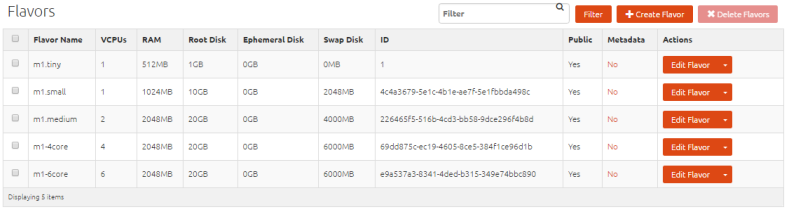


Gambar 4.5: Tampilan Daftar Layanan Nova yang telah berjalan

Pengujian selanjutnya ialah melakukan konfigurasi terhadap *flavor* dari mesin virtual. *Flavor* merupakan pilihan spesifikasi mesin virtual yang dapat digunakan oleh pengguna. *Flavor* dapat disesuaikan dengan kebutuhan dari pengguna, selama besar spesifikasi yang ditawarkan tidak melebihi kapasitas dari seluruh *compute node* yang ada. Tujuan dari pengujian ini adalah untuk menyesuaikan besar *flavor* yang akan digunakan terhadap kapasitas dari kedua *compute node* yang digunakan. Hal ini perlu diperhatikan, karena kedua *compute node* memiliki jumlah CPU *core* yang berbeda, serta besar memori yang tidak terlalu banyak (4GB), sehingga dapat memengaruhi besar spesifikasi dan jumlah mesin virtual yang dapat berjalan pada kedua *compute node*.

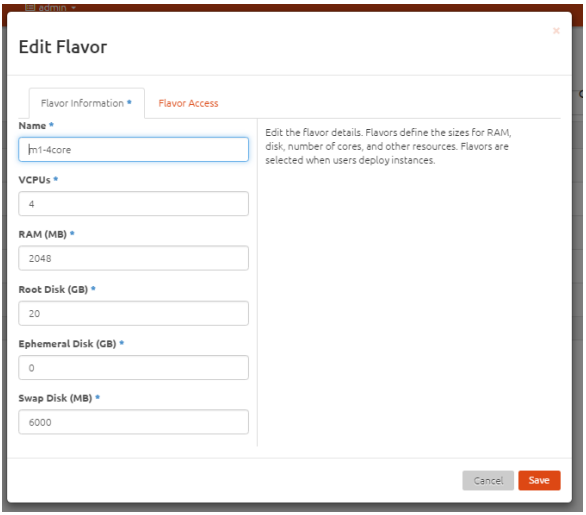
Dengan menggunakan akun administrator melalui Horizon, pengaturan *flavor* dilakukan dengan memilih pilihan **Flavors** pada menu **Admin - System**. Pada halaman tersebut, akan terlihat seluruh *flavor* yang telah tersedia (gambar 4.6). Terdapat pilihan untuk membuat *flavor* baru, menghapus *flavor* yang tersedia, serta mengubah spesifikasi dari *flavor* yang telah ada.

Pada penelitian ini, dibutuhkan dua *Flavor* yang akan digunakan pada pengujian mesin virtual, masing-masing dibedakan oleh jumlah CPU *core* untuk memaksimalkan kapasitas dari masing-masing *compute node*. Spesifikasi dari kedua *flavor* tersebut dijabarkan pada tabel 4.1 dan 4.2. Untuk membuat sebuah *flavor* maka



Gambar 4.6: Tampilan Daftar *Flavors* yang tersedia

pilih **Create Flavor** untuk menampilkan formulir pembuatan *flavor* baru. Pembuatan *flavor* baru juga dapat dilakukan dengan mengubah *flavor* yang telah ada dengan menyesuaikan spesifikasi *flavor* dengan yang kita inginkan (gambar 4.7).



Gambar 4.7: Tampilan Perubahan Spesifikasi *flavor*

Pengujian selanjutnya adalah mengalokasikan jaringan publik yang nantinya akan digunakan pada mesin virtual agar dapat diakses dari luar. Pembagian alamat IP menggunakan metode DHCP (*Dynamic Host Configuration Protocol*) yang telah disediakan oleh nova-network. Tidak seperti berbagai layanan lain dari OpenStack,

Tabel 4.1: Spesifikasi *flavor* "m1.4core"

Spesifikasi	Keterangan
Jumlah Prosesor	4
Memori	2 GB
Disk Storage	20 GB
Swap Disk Size	6 GB

Tabel 4.2: Spesifikasi *flavor* "m1.6core"

Spesifikasi	Keterangan
Jumlah Prosesor	6
Memori	2 GB
Disk Storage	20 GB
Swap Disk Size	6 GB

pengalokasian alamat IP hanya dapat dilakukan secara akses langsung pada *compute node*. Selain itu, perintah tersebut hanya dapat dilakukan oleh akun administrator. Dalam menggunakan akun administrator pada *console*, maka identitas akun perlu dimasukan pada *Environment Variables* milik sistem terlebih dahulu, agar perintah-perintah dari layanan OpenStack dapat dijalankan (gambar 4.1).

```
export OS_TENANT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=keystone_admin
export OS_AUTH_URL=http://controller:35357/v2.0
```

Kode 4.1: Perintah Ekspor Identitas Administrator

Setelah perintah-perintah tersebut dijalankan, maka administrator dapat menjalankan perintah-perintah administratif dari OpenStack melalui *console* tersebut. Untuk melakukan pengecekan terhadap alokasi jaringan untuk mesin virtual, dijalankan perintah `nova net-list` untuk menampilkan seluruh jaringan yang telah dialokasikan. Pada penelitian ini, seluruh mesin virtual dialokasikan pada jaringan IP publik 10.122.1.0/24 (gambar 4.8) dengan IP pertama

dimulai pada 10.122.1.234.

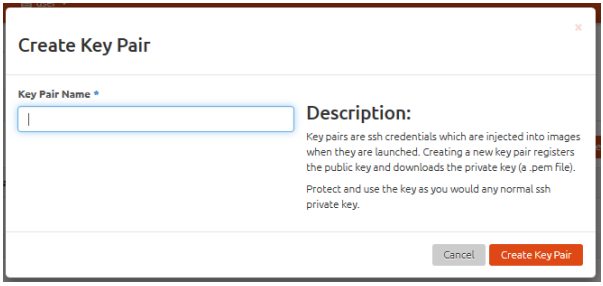
```
root@controller:~/keystone_script# nova net-list
+--------------------------------------+-------+---------------+
| ID                                   | Label | CIDR          |
+--------------------------------------+-------+---------------+
| 2ed7dbc3-1e3d-4ee4-af92-5716b8fd1394 | vmnet | 10.122.1.0/24 |
+--------------------------------------+-------+---------------+
```

Gambar 4.8: Daftar jaringan yang teralokasikan pada Nova-network

4.1.4 Pembuatan Mesin Virtual

Setelah semua layanan OpenStack berfungsi dengan baik, maka pengujian dilanjutkan dengan melakukan pembuatan mesin virtual menggunakan akun pengguna biasa. Pada bagian ini, pengujian dilakukan menggunakan akun pengguna biasa, sehingga berbagai perintah administratif tidak akan dijalankan.

Sebelum melakukan pembuatan mesin virtual, pengguna diharuskan untuk membuat *key pair* terlebih dahulu. *Key pair* tersebut digunakan untuk mengakses mesin virtual yang telah dibuat. Pembuatan *Key pair* dilakukan melalui menu **Project - Compute**, kemudian pada menu **Access and Security** pada tab **Key Pairs** pilih **Create Key Pair**, maka akan tampil halaman pembuatan *key pair* (gambar 4.9).

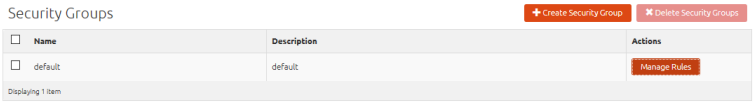


Gambar 4.9: Halaman pembuatan *key pair* baru

Masukan nama dari *key pair* yang diinginkan, kemudian tekan **Create Key Pair** maka *public key* secara otomatis telah didaftarkan pada OpenStack. *Private key* dari *key pair* yang baru saja

dibuat akan terunduh secara otomatis pada *web browser* pengguna. Perlu diperhatikan bahwa *key pair* yang telah diunduh harus disimpan dengan baik, karena tidak dapat diunduh ulang.

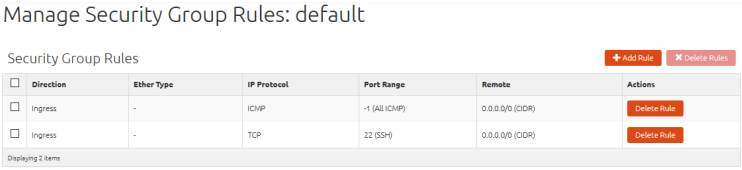
Langkah selanjutnya adalah mengatur konfigurasi keamanan *security* dari mesin virtual. Pada halaman yang sama, pilih tab **Security Groups**, maka pengguna akan dibawa pada daftar *security groups* yang ada (gambar 4.10).



Gambar 4.10: Daftar *Security Groups* milik pengguna

Secara *default*, seluruh akses *port* pada mesin virtual ditutup oleh OpenStack, sehingga agar mesin virtual dapat diakses dari luar, *port-port* yang dibutuhkan oleh pengguna harus ditambahkan pada *security groups* terlebih dahulu. Pada *Security Groups* yang tersedia, pilih tombol **Manage Rules** dan tambahkan berbagai protokol serta *port* yang pengguna inginkan pada daftar *rules* yang tersedia pada *security group* tersebut (gambar 4.11).

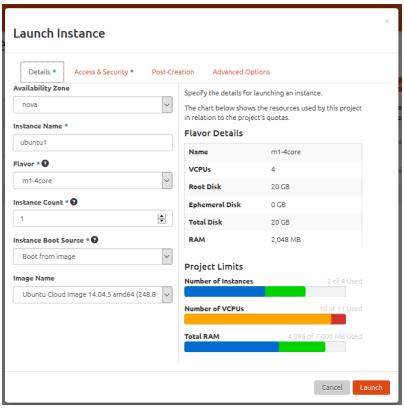
Beberapa *rules* yang perlu ditambahkan diantaranya protokol ICMP (*Internet Control Message Protocol*) yang digunakan untuk melakukan *Ping* terhadap mesin virtual, serta TCP (*Transmission Control Protocol*) *port* 22, yang digunakan untuk melakukan akses mesin virtual melalui SSH (*Secure Shell*).



Gambar 4.11: Daftar *Rules* pada *Security Groups*

Setelah seluruh pengaturan awal tersebut telah dilakukan, pengguna dapat langsung membuat mesin virtual. Pembuatan mesin virtual dapat dilakukan pada halaman **Project - Compute**, lalu pada menu **Instances** tekan tombol **Launch Instance**, maka pengguna

akan dibawa pada formulir pembuatan mesin virtual seperti pada gambar 4.12.

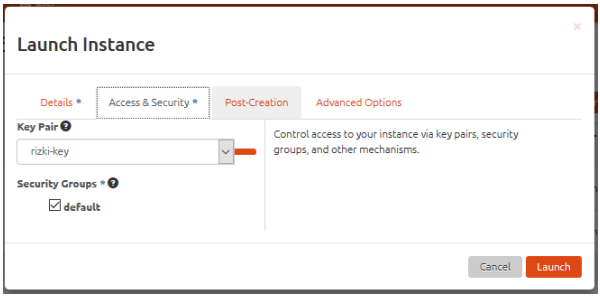


Gambar 4.12: Halaman pembuatan mesin virtual baru

Pada formulir tersebut, tentukan nama mesin virtual (*Instance Name*), jenis *flavor* yang diinginkan, jumlah mesin virtual, serta pilihan *boot image* dari sistem operasi yang diinginkan. Pada tab **Access and Security**, pengguna juga harus menentukan *key pair* yang akan digunakan pada mesin virtual serta jenis *security group* yang akan digunakan, seperti pada gambar 4.13. Setelah seluruh parameter telah dimasukkan, maka pengguna tinggal menekan tombol **Launch** agar mesin virtual baru dapat dibuat.

Untuk keperluan pengujian, digunakan dua buah mesin virtual dengan spesifikasi berbeda, yang telah dijabarkan pada tabel 4.1 dan 4.2 pada pengujian sebelumnya. Tujuannya agar kedua mesin virtual tersebut dapat menggunakan seluruh CPU *core* yang tersedia pada kedua *compute node*, agar masing-masing dari kedua mesin virtual tersebut berjalan pada kedua *compute node* yang digunakan. Untuk mempermudah identifikasi mesin virtual pada pengujian selanjutnya, maka kedua mesin virtual diberi nama **ubuntu1** untuk mesin virtual dengan konfigurasi empat CPU, serta **ubuntu2** untuk mesin virtual dengan konfigurasi enam CPU.

Proses pembentukan mesin virtual baru membutuhkan waktu beberapa detik hingga beberapa menit, bergantung dari performa



Gambar 4.13: Pengaturan Akses dan Keamanan pada pembuatan Mesin Virtual

dari *compute node* yang digunakan. Setelah terbentuk, mesin virtual tersebut akan tampak pada daftar **Instances** milik pengguna, seperti yang dicontohkan pada gambar 4.14.

Gambar 4.14: Daftar mesin virtual milik pengguna

Setelah mesin virtual telah terbentuk, maka disarankan untuk melakukan *ping* terhadap kedua mesin virtual yang telah dibuat seperti pada gambar 4.15 serta 4.16, sebagai bentuk konfirmasi bahwa mesin virtual telah terhubung dengan jaringan publik dan pengaturan pada *security group* telah berjalan sesuai dengan yang telah dikonfigurasikan. *Ping* dilakukan terhadap kedua mesin virtual dan kedua melalui alamat IP dari masing-masing mesin virtual, yakni 10.122.1.234 dan 10.122.1.235.

Setelah kedua mesin virtual telah berjalan dengan baik, maka pengguna dapat melakukan koneksi jarak jauh *remote* menggunakan SSH dengan menyertakan *key pair* yang telah dibuat dan diunduh sebelumnya.

```
C:\Users\rb2>ping 10.122.1.234

Pinging 10.122.1.234 with 32 bytes of data:
Reply from 10.122.1.234: bytes=32 time=1ms TTL=64
Reply from 10.122.1.234: bytes=32 time<1ms TTL=64
Reply from 10.122.1.234: bytes=32 time<1ms TTL=64
Reply from 10.122.1.234: bytes=32 time=1ms TTL=64

Ping statistics for 10.122.1.234:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

Gambar 4.15: Proses *ping* terhadap mesin virtual pertama

```
C:\Users\rb2>ping 10.122.1.235

Pinging 10.122.1.235 with 32 bytes of data:
Reply from 10.122.1.235: bytes=32 time=1ms TTL=64
Reply from 10.122.1.235: bytes=32 time=1ms TTL=64
Reply from 10.122.1.235: bytes=32 time<1ms TTL=64
Reply from 10.122.1.235: bytes=32 time<1ms TTL=64

Ping statistics for 10.122.1.235:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 1ms, Average = 0ms
```

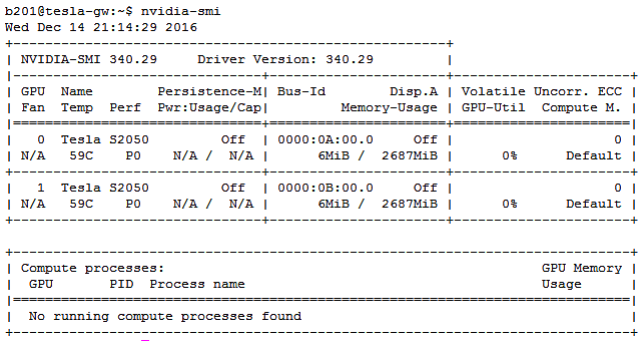
Gambar 4.16: Proses *ping* terhadap mesin virtual kedua

4.2 Pengujian GPU *Server*

4.2.1 Pemeriksaan GPU *Driver*

Tahap pengujian berikutnya adalah menguji layanan dan perangkat keras dari GPU *server*. Tujuan utama dari pengujian ini adalah untuk memeriksa kondisi dari GPU yang digunakan, menguji layanan rCUDA *server* serta *library* milik CUDA *toolkit*, serta untuk mendapatkan parameter performa yang akan digunakan untuk pengujian komputasi GPU pada bab berikutnya.

Pengujian pertama yang dilakukan adalah memeriksa apakah *driver* dari GPU telah terpasang dan dapat mendeteksi GPU yang digunakan. Versi *driver* yang digunakan termasuk dalam paket instalasi CUDA *toolkit* versi 6.5 yang digunakan pada GPU *server*. Untuk mendeteksi GPU yang telah terpasang, dapat digunakan sebuah program bawaan dari *driver*, yakni NVIDIA SMI (*System Ma-*



Gambar 4.17: Tampilan output NVIDIA SMI

nagement Interface). Untuk menjalankan program tersebut, melalui *console* pada GPU *server* ketik `nvidia-smi`, maka selanjutnya akan tampil output dari program yang menampilkan kondisi dari GPU saat program dijalankan (gambar 4.17).

Pada output program, akan ditampilkan seluruh GPU yang dapat dideteksi pada GPU *server*. Pada pengujian ini, `nvidia-smi` hanya dapat mendeteksi dua dari keempat GPU yang ada pada modul GPU NVIDIA Tesla. Hal ini dikarenakan dua GPU lainnya sangat tidak stabil saat dibaca oleh `nvidia-smi` serta pada saat digunakan untuk melakukan proses komputasi GPU. Pada saat seluruh GPU digunakan, akan sering terjadi *soft lockup* secara terus menerus pada *kernel* sistem operasi milik GPU *server* sehingga menyebabkan stabilitas dari sistem terganggu, seperti pada kode 4.2

```
Oct 18 13:22:33 tesla-gw kernel: [152580.100003] BUG: soft
    lockup - CPU#1 stuck for 23s! [nvidia-smi:7889]
Oct 18 13:22:56 tesla-gw kernel: [152580.100003] BUG: soft
    lockup - CPU#1 stuck for 23s! [nvidia-smi:7889]
Oct 18 13:23:19 tesla-gw kernel: [152580.100003] BUG: soft
    lockup - CPU#1 stuck for 23s! [nvidia-smi:7889]
Oct 18 13:22:42 tesla-gw kernel: [152580.100003] BUG: soft
    lockup - CPU#1 stuck for 23s! [nvidia-smi:7889]
```

Kode 4.2: Galat pada *kernel* saat pengujian GPU

Salah satu solusi yang telah dilakukan yaitu dengan melepas kabel interkoneksi milik kedua GPU yang tidak stabil tersebut dari

Host Interface Card. Solusi ini dapat memecahkan masalah stabilitas pada sistem saat melakukan komputasi menggunakan GPU, namun mengurangi jumlah GPU yang digunakan dari empat buah menjadi hanya dua buah.

Program `nvidia-smi` juga dapat memperlihatkan tingkat utilisasi dari masing-masing GPU, serta seluruh proses komputasi yang sedang menggunakan GPU tersebut.

4.2.2 Pemeriksaan Instalasi CUDA *Toolkit*

Pengujian selanjutnya adalah memeriksa instalasi *library* dari CUDA *toolkit*. *Library* dari NVIDIA CUDA dibutuhkan agar program komputasi yang menggunakan CUDA dapat dikompilasi dijalankan oleh sistem. Pengujian ini dilakukan dengan menjalankan beberapa kode sampel yang disertakan pada saat instalasi CUDA *toolkit*. Sebagian dari kode sampel tersebut juga dapat dilakukan untuk melakukan pengujian kinerja yang akan dilakukan pada bagian selanjutnya.

Salah satu kode sampel yang akan dijalankan yaitu `deviceQuery`. Program ini memiliki fungsi yang cukup sederhana, yaitu membaca beberapa parameter teknis yang dimiliki dari setiap GPU yang terdeteksi oleh *driver*. Program terlebih dahulu harus dikompilasi sebelum dijalankan. Untuk mengompilasi program, cukup dengan menjalankan perintah `make` pada direktori tempat kode program diletakkan. Apabila program dapat berjalan, maka *library* CUDA *toolkit* telah berjalan dengan baik. Ketika program akan mendeteksi seluruh GPU yang terpasang pada GPU *server* dan menampilkan seluruh data teknis dari semua GPU yang ada tersebut. Output dari `deviceQuery` ditampilkan pada kode 6.1 pada bagian lampiran.

4.2.3 Pemeriksaan Instalasi RCUDA *Server*

Pengujian selanjutnya adalah menjalankan *server daemon* dari rCUDA, agar mesin virtual dapat berkomunikasi dengan GPU *server* untuk melakukan virtualisasi jarak jauh.

Untuk menjalankan *server*, maka cukup dengan menjalankan program *server daemon* milik rCUDA yang bernama `rCUDAd`, maka secara otomatis program tersebut akan berjalan sebagai *background program*. Output dari program tersebut ditampilkan pada gambar

4.18.

```
b201@tesla-gw:~/rCUDAv15.07-CUDA6.5/bin$ ./rCUDAd
rCUDAd v15.07
Copyright 2009-2015 UNIVERSITAT POLITECNICA DE VALENCIA. All rights reserved.
'rCUDAd' started in background.
b201@tesla-gw:~/rCUDAv15.07-CUDA6.5/bin$
```

Gambar 4.18: Tampilan output program `rCUDAd`

Untuk melihat nomor PID (*Process Identifier Number*) dari `rCUDAd`, dapat menggunakan perintah "`ps ax | grep rCUDA`" sehingga akan ditampilkan seluruh *process* dengan nama "`rCUDA`" seperti pada gambar 4.19.

```
b201@tesla-gw:~$ ps ax | grep rCUDA
 6062 ?        Ss     0:00 ./rCUDAd
 6066 ?        Sl     0:00 ./rCUDAd
 6105 pts/1    S+     0:00 grep --color=auto rCUDA
b201@tesla-gw:~$
```

Gambar 4.19: Perintah untuk menampilkan PID dari proses

Terdapat dua *process* yang berjalan saat *server daemon* dijalankan, masing-masing memiliki fungsi yang berbeda meskipun memiliki nama *process* yang sama. Salah satu *process* bertugas menerima *request* dari *remote client* untuk melakukan virtualisasi GPU, sementara *process* lainnya bertugas mengeksekusi kode program CUDA yang dijalankan pada *remote client* menggunakan GPU, dan terhubung secara langsung dengan perangkat GPU yang tersedia pada *server*.

Untuk menghentikan *server daemon*, maka *process* harus dihentikan dengan menggunakan perintah `kill` diikuti dengan nomor PID dari *process* yang akan dihentikan. Setelah *process* dihentikan, maka perintah "`ps ax | grep rCUDA`" dapat digunakan untuk memastikan seluruh *process* milik `rCUDAd` sudah tidak berjalan.

Nomor PID dari *server daemon* tersebut juga dapat dilihat melalui program `nvidia-smi` seperti pada gambar 4.20. Tidak seperti menggunakan perintah "`ps ax | grep rCUDA`", perintah `nvidia-smi` hanya dapat menampilkan proses yang sedang terhubung dengan GPU secara langsung, sehingga hanya dapat menampilkan satu *process* `rCUDAd`.

```
b201@tesla-gw:~$ nvidia-smi
Wed Dec 14 22:57:24 2016
+------------------------------------------------------+
| NVIDIA-SMI 340.29     Driver Version: 340.29         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla S2050         Off  | 0000:0A:00.0     Off |                    0 |
| N/A   62C    P0    N/A /  N/A |     55MiB /  2687MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla S2050         Off  | 0000:0B:00.0     Off |                    0 |
| N/A   62C    P1    N/A /  N/A |      6MiB /  2687MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Compute processes:                                               GPU Memory |
|  GPU       PID  Process name                                     Usage      |
|=============================================================================|
|    0      6066  ./rCUDAd                                            47MiB |
+-----------------------------------------------------------------------------+
```

Gambar 4.20: Tampilan nomor PID melalui *nvidia-smi*

Program `rCUDAd` juga dapat dijalankan sebagai program interaktif, bukan sebagai *background process* atau *daemon*. Dengan menggunakan mode ini, program `rCUDAd` dapat dihentikan secara langsung sewaktu-waktu tanpa menggunakan perintah `kill`. Selain itu, pada mode ini, seluruh *request* dari *client* akan tertulis pada output program. Contoh dari program `rCUDAd` yang dijalankan pada mode interaktif ditampilkan pada gambar 4.21.

```
b201@tesla-gw:~/rCUDAv15.07-CUDA6.5/bin$ ./rCUDAd -i
rCUDAd v15.07
Copyright 2009-2015 UNIVERSITAT POLITECNICA DE VALENCIA. All rights reserved.
rCUDAd[6443]: Using rCUDAcommTCP.so communications library.
rCUDAd[6443]: Server daemon succesfully started.
```

Gambar 4.21: Tampilan program `rCUDAd` pada mode interaktif

4.3 Pengujian Virtualisasi GPU pada Mesin Virtual

Pengujian selanjutnya adalah memeriksa instalasi dari CUDA *toolkit* serta rCUDA *library* pada kedua mesin virtual. Pengujian yang dilakukan hampir serupa dengan pengujian pada GPU *server*, yakni dengan menjalankan sampel program dari CUDA *toolkit*. Perbedaan yang paling mendasar antara pengujian pada GPU *server* dengan pengujian pada mesin virtual atau *remote client* adalah pada *remote client* konfigurasi dari *compiler* yang digunakan pada

proses kompilasi kode program harus disesuaikan agar *library* milik rCUDA juga ikut disertakan pada proses kompilasi.

```
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

mlock error: Cannot allocate memory
[rCUDA-280] rCUDA warning: Buffer's memory cannot be locked (2048KB). Please che
ck the memlock limit in your system (limits.conf man page)
cudaGetDeviceCount returned 30
-> unknown error
Result = FAIL
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$
```

Gambar 4.22: Galat pada program CUDA

Secara *default*, seluruh kode program CUDA dikompilasi dengan menggunakan *static-linking*, dimana beberapa bagian dari *library* yang digunakan pada kode diikutsertakan menjadi bagian dari program. Hal ini akan menyebabkan galat (*error*) pada program, karena CUDA tidak mendeteksi adanya perangkat GPU pada sistem (gambar 4.22).

RCUDA mengharuskan kode program yang akan dijalankan untuk dikompilasi menggunakan *dynamic linking* agar *library* milik rCUDA dapat terbaca oleh program setelah dikompilasi. Hal ini dapat dilakukan dengan menambahkan sebuah *compiler flag* yang mewajibkan *compiler* untuk menggunakan *dynamic linking* pada proses kompilasi program.

Langkah pertama yang dilakukan yaitu dengan menambahkan letak *library* dari rCUDA pada daftar *library* milik sistem. Dengan menjalankan perintah pada kode 4.3, maka *library* telah berhasil didaftarkan pada sistem.

```
export LD_LIBRARY_PATH="/usr/local/rCUDA/lib/"
```

Kode 4.3: Perintah untuk menambahkan *library* rCUDA pada sistem

Pada kode sampel program CUDA, *dynamic linking* dapat dilakukan dengan menambahkan dua *compiler flag* (tertulis pada kode 4.4) pada Makefile dari program tersebut.

```
...
# Extra user flags
```

```
EXTRA_NVCCFLAGS   ?= -cudart=shared #untuk compiler NVCC
EXTRA_LDFLAGS   ?=
EXTRA_CCFLAGS   ?= -lcudart #untuk compiler GCC / G++
...
```

Kode 4.4: *Compiler flag* untuk melakukan *dynamic linking*

Untuk memeriksa apakah program tersebut menggunakan *library* dari rCUDA, maka program tersebut dapat diperiksa menggunakan perintah `ldd`. `Ldd` menampilkan seluruh *dynamic library* yang digunakan oleh program pada proses kompilasi. Output dari `ldd` terhadap program `deviceQuery` ditampilkan pada gambar 4.23. Apabila salah satu *library* yang digunakan oleh program merupakan *library* milik rCUDA, maka program dapat dijalankan menggunakan virtualisasi GPU.

```
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$ ldd deviceQuery
        linux-vdso.so.1 =>  (0x00007ffcbe9a4000)
        libcudart.so.6.5 => /usr/local/rCUDA/lib/libcudart.so.6.5 (0x00007f5181137000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f5180e33000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f5180c1d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5180858000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f518063a000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f5180436000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f5180130000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f51813ca000)
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$
```

Gambar 4.23: Output dari program `ldd` terhadap `deviceQuery`

Langkah selanjutnya adalah menambahkan beberapa *Environment Variables* pada sistem. Isi dari variabel-variabel tersebut adalah alamat IP dari seluruh rCUDA *server* yang ada pada jaringan, serta jumlah GPU *server* yang dapat diakses oleh *remote client* tersebut. *Environment Variables* yang perlu ditambahkan pada mesin virtual dalam penelitian ini ditulis pada kode 4.5.

```
RCUDA_DEVICE_COUNT=2      #Jumlah GPU Virtual yang digunakan
RCUDA_DEVICE_0=10.122.1.250   #Alamat IP GPU virtual 1
RCUDA_DEVICE_1=10.122.1.250:1 #Alamat IP GPU virtual 2
```

Kode 4.5: *Environment Variables* yang ditambahkan pada *Remote Client*

Apabila *Environment Variables* tersebut tidak ditambahkan, maka program akan mengeluarkan galat *error* yang memberitahukan untuk menambahkan *Environment Variables* yang sesuai, seperti pada gambar 4.24.

```
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

[rCUDA-642] rCUDA error: rCUDA: Please, set the RCUDA_DEVICE_COUNT environment v
ariable with the number of devices accessible from this host.
ubuntu@ubuntu1:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$
```

Gambar 4.24: Galat pada program akibat *Environment Variables* tidak ditambahkan

Setelah semua pengaturan diatas telah selesai dilakukan dan program `deviceQuery` telah dikompilasi, maka program tersebut dapat dijalankan dan mengakses informasi dari GPU milik *server*. Apabila output dari `deviceQuery` pada mesin virtual sama dengan output pada GPU *server*, maka seluruh instalasi CUDA *toolkit* dan rCUDA *library* telah berhasil dan dapat digunakan.

4.4 Pengujian Komputasi GPU

Tahap terakhir dari rangkaian pengujian adalah pengukuran performansi komputasi berbasis GPU. Pengujian ini dilakukan pada kedua mesin virtual, **ubuntu1** (4 *core*), **ubuntu2** (6 *core*), serta pada GPU *server*. Tujuan dari pengujian ini adalah untuk mendapatkan gambaran kinerja komputasi GPU menggunakan metode virtualisasi jarak jauh dengan non-virtualisasi (konvensional).

Pengujian pertama yang dilakukan adalah dengan menjalankan program sampel CUDA bernama `bandwidthTest`. Program ini memiliki fungsi mengukur besar *bandwidth* antara memori utama milik sistem (*host memory*) dengan memori milik GPU. Tujuan dari pengukuran ini adalah untuk mengetahui besar pengaruh *bandwidth* dari jaringan yang digunakan pada sistem terhadap penggunaan GPU secara virtualisasi. Pengujian dilakukan pada GPU *server* secara langsung serta pada mesin virtual melalui virtualisasi. Program ini telah disertakan pada CUDA *toolkit*.

Pengujian kedua yang dilakukan adalah melakukan kalkulasi perkalian dua buah matriks berukuran 1024 x 1024. Data pertama yang diambil adalah kalkulasi menggunakan CPU dengan jumlah *core* yang berbeda, sementara data kedua berasal dari hasil kalkulasi menggunakan GPU, baik secara lokal pada GPU *server* maupun secara virtualisasi jarak jauh pada kedua mesin virtual, untuk men-

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla S2050
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     113.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     117.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     108814.6

Result = PASS
```

Gambar 4.25: Output dari program `bandwidthTest`

dapatkan gambaran perbedaan performa komputasi pada berbagai macam spesifikasi.

Untuk melakukan kalkulasi perkalian matriks pada GPU, digunakan program sampel dari CUDA *toolkit* bernama `matrixMul`. Program sampel tersebut berfungsi sebagai alat demonstrasi pembuatan kode program berbasis CUDA, sehingga tidak memberikan performa kalkulasi yang sangat optimal, namun lebih dari cukup untuk dapat digunakan sebagai tolak ukur performa yang dilakukan pada penelitian ini. Kekurangan dari program ini adalah program hanya dapat menggunakan sebuah GPU untuk melakukan kalkulasi kedua matriks.

Secara *default*, `matrixMul` hanya dapat melakukan perkalian matriks berukuran hingga 640 x 640, sehingga diperlukan sedikit penyesuaian pada kode program agar secara *default* ketika program dijalankan, program langsung melakukan kalkulasi dua matriks berukuran 1024 x 1024. Output dari program `matrixMul` dilampirkan pada gambar 4.26

Untuk melakukan kalkulasi perkalian matriks pada CPU, maka diperlukan sebuah program tersendiri yang ditulis untuk melakukan kalkulasi perkalian matriks secara *multithreaded* menggunakan POSIX *threads* (pthread). Tujuannya agar kalkulasi matriks da-

```
b201@tesla-gw:~/NVIDIA_CUDA-6.5_Samples/0_Simple/matrixMul$ cd ~
b201@tesla-gw:~$ ./matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla S2050" with compute capability 2.0

MatrixA(1024,1024), MatrixB(1024,1024)
Computing result using CUDA Kernel...
done
Performance= 178.33 GFlop/s, Time= 12.042 msec, Size= 2147483648 Ops, WorkgroupS
ize= 1024 threads/block
Checking computed result for correctness: Result = PASS

Note: For peak performance, please refer to the matrixMulCUBLAS example.
```

Gambar 4.26: Contoh Output dari program `matrixMul`

pat dilakukan menggunakan jumlah *core* yang berbeda-beda. Kode dari program bernama `matrix.c` tersebut ditulis pada kode 6.2 pada bab 6. Program tersebut dijalankan dengan menambahkan argumen berupa jumlah CPU *core* yang digunakan dalam proses kalkulasi (`./matrix [jumlah core]`).

Pengujian komputasi CPU dilakukan dengan menggunakan jumlah CPU *core* yang bervariasi, dari satu *core* hingga enam *core*, sesuai dari spesifikasi CPU pada GPU *server* dan mesin virtual yang telah dibuat. Tujuannya untuk mengamati peningkatan performa komputasi seiring dengan penambahan CPU *core*.

4.4.1 Pengujian pada GPU *server*

Pengujian komputasi dimulai dengan melakukan pengukuran *bandwidth* antara memori sistem dengan memori GPU menggunakan `bandwidthTest`. Program dijalankan secara langsung pada GPU *server* sebagai bentuk pengujian terhadap penggunaan GPU secara langsung tanpa virtualisasi. Data yang diambil akan digunakan sebagai tolak ukur terhadap pengukuran Dari data pengu-

Tabel 4.3: Hasil pengukuran `bandwidthTest` pada GPU *server*

Jenis Transfer	***Bandwidth*** **(MB/s)**
Host to Device	3003 MB/s
Device to Host	3291 MB/s
Device to Device	109729 MB/s

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla S2050
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     3003.0

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     3291.2

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     109729.4

Result = PASS
```

Gambar 4.27: Output dari program `bandwidthTest` pada GPU *server*

kuran *bandwidth* diatas, tampak

Pengujian selanjutnya yaitu melakukan komputasi perkalian matriks dengan menjalankan program `matrixMul` serta `matrix.c` pada GPU *server* untuk mendapatkan tolak ukur performa komputasi pada lingkungan non virtualisasi (*bare-metal system*) yang nantinya akan dibandingkan dengan hasil pengujian dari kedua mesin virtual. Untuk membatasi perbedaan yang terlalu besar, maka kecepatan CPU pada GPU *server* diturunkan dari 2.66 GHz menjadi 1.86 GHz agar dapat menyamai kecepatan CPU *core* yang digunakan pada kedua *compute node*.

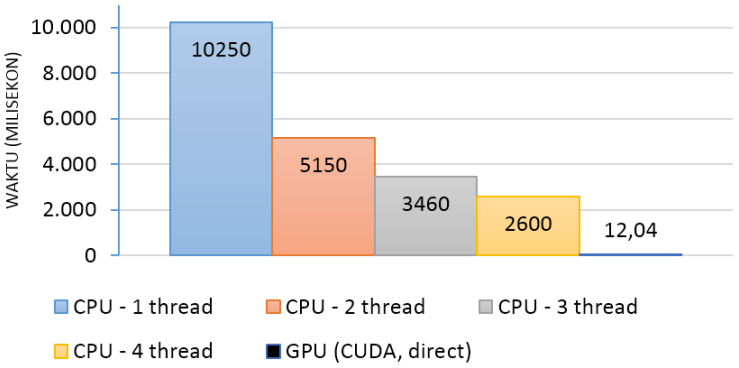
Tabel 4.4: Spesifikasi GPU *server* pada Pengujian Komputasi

Spesifikasi	Keterangan
Jumlah Prosesor	4
Kecepatan Prosesor	1.86 GHz
Metode Komputasi GPU	CUDA secara langsung

Tolak ukur yang digunakan pada pengujian ini adalah waktu yang dibutuhkan untuk menyelesaikan proses komputasi masing-

Tabel 4.5: Waktu Kalkulasi Perkalian Matriks pada GPU *server*

Jumlah CPU *Core*	**Waktu (milisekon)**
1 *Core*	10250 ms
2 *Core*	5150 ms
3 *Core*	3460 ms
4 *Core*	2600 ms
GPU (CUDA)	12,04 ms



Gambar 4.28: Grafik perbedaan waktu pemrosesan perkalian matriks pada GPU *server*

masing dari kedua program tersebut, dimana semakin cepat proses komputasi selesai dilakukan, semakin baik performa komputasi dari sistem tersebut.

Pada tabel 4.5 diperlihatkan waktu yang dibutuhkan oleh GPU *server* dalam menyelesaikan perkalian dua buah matriks berukuran 1024 x 1024 dengan menggunakan berbagai konfigurasi CPU *core* yang ada, serta dengan menggunakan program CUDA khusus yang menggunakan kemampuan dari GPU. Dari tabel tersebut, dengan hanya menggunakan sebuah CPU *core*, proses kalkulasi membutuhkan waktu selama 10,25 detik (10250 milisekon). Penambahan

sebuah *core* dapat mengurangi waktu yang dibutuhkan hingga separuhnya menjadi 5,15 detik (5150 milisekon). Dengan menambahkan dua buah *core* lagi maka waktu yang dibutuhkan kembali berkurang sebesar separuhnya menjadi 2,6 detik (2600 milisekon). Dari data-data tersebut dapat ditarik kesimpulan bahwa penambahan CPU *core* sebanyak dua kali dari jumlah awal dapat mengurangi waktu yang dibutuhkan dalam proses komputasi hingga separuh.

Pengujian juga dilakukan terhadap GPU dengan menggunakan program `matrixMul` berbasis *library* CUDA. Dari waktu yang didapatkan dari output program seperti pada gambar 4.26, perkalian matriks menggunakan GPU hanya membutuhkan waktu jauh di bawah 1 detik, pada fraksi sekian milisekon (12,04 milisekon). Angka tersebut terpaut sangat jauh dari waktu yang dibutuhkan untuk melakukan tugas yang serupa menggunakan empat buah CPU *core*, seperti yang digambarkan pada grafik 4.28. Hal ini menunjukkan penggunaan GPU yang sangat efisien dalam melakukan perintah kalkulasi serta iterasi yang panjang. Karena jumlah GPU *core* yang mencapai 448 CUDA *core*, setiap iterasi perkalian matriks dapat dibagikan pada setiap GPU *core* tersebut, sehingga proses iterasi berjalan dengan sangat cepat.

4.4.2 Pengujian pada Mesin Virtual

Pengujian komputasi selanjutnya dilakukan pada dua buah mesin virtual yang telah dibuat. Tujuan dari bagian pengujian ini adalah untuk mendapatkan gambaran performa komputasi dengan menggunakan mesin virtual serta virtualisasi GPU. Pengujian yang dilakukan serupa dengan pengujian yang dilakukan pada GPU *server* dengan beberapa penyesuaian pada proses kompilasi program agar dapat berjalan menggunakan virtualisasi GPU.

Mesin virtual pertama yang diuji adalah **ubuntu1**, yang berjalan pada *Compute Node 1*. Spesifikasi dari *Compute Node* tersebut dapat dilihat pada tabel 3.2 pada Bab 3. Hal ini perlu diperhatikan, karena perangkat keras yang digunakan sebagai *Compute Node* pertama tersebut jauh lebih tua dari perangkat keras yang digunakan pada GPU *server* serta *Compute Node* kedua.

Seperti pada yang diperlihatkan pada tabel 4.7, dengan hanya menggunakan satu CPU *core*, sistem membutuhkan waktu selama

Tabel 4.6: Spesifikasi Mesin Virtual **Ubuntu1** pada Pengujian Komputasi

Spesifikasi	**Keterangan**
Jumlah Prosesor	4
Kecepatan Prosesor	1.86 GHz
Metode Komputasi GPU	CUDA secara virtualisasi menggunakan RCUDA

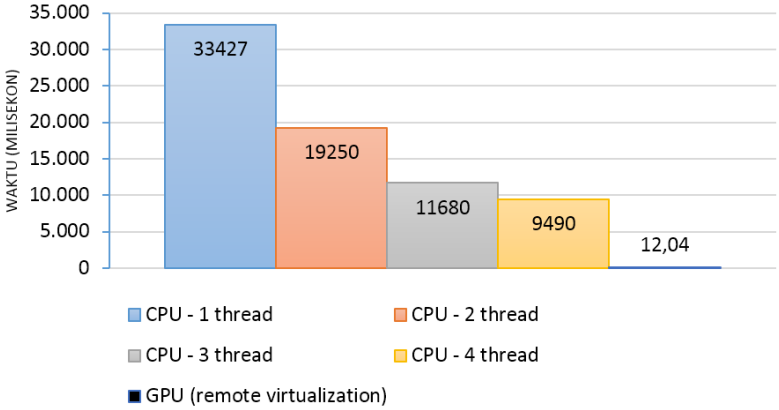
Tabel 4.7: Waktu Kalkulasi Perkalian Matriks pada Mesin Virtual **Ubuntu1**

Jumlah CPU *Core*	**Waktu (milisekon)**
1 *Core*	33427 ms
2 *Core*	19250 ms
3 *Core*	11680 ms
4 *Core*	9490 ms
GPU (CUDA)	12,04 ms

hampir 35 detik (33427 milisekon) untuk menyelesaikan perkalian matriks. Dengan menambahkan satu CPU *core* maka waktu yang dibutuhkan berkurang menjadi 19 detik (19250 milisekon). Penggunaan empat buah CPU *core* menyebabkan waktu yang dibutuhkan berkurang menjadi hampir 10 detik (9490 milisekon).

Tren ini sama seperti pada GPU *server*, dimana penambahan CPU *core* sebanyak dua kali lipat dapat mengurangi waktu yang dibutuhkan hingga separuhnya. Pengecualian terdapat pada penggunaan dua buah CPU *core*, dimana pengurangan waktu pengerjaan perkalian matriks hanya berkurang sebesar 42 persen dari waktu saat menggunakan sebuah CPU *core*.

Pengujian komputasi menggunakan GPU dilakukan dengan metode yang sama dengan pengujian pada GPU *server*. Perbedaan utama dari pengujian sebelumnya adalah program dikompilasi dan dijalankan menggunakan *library* rCUDA, agar program dapat mendeteksi GPU virtual pada jaringan. Dari tabel 4.6, perkali-



Gambar 4.29: Grafik perbedaan waktu pemrosesan perkalian matriks pada *Ubuntu1*

an matriks menggunakan GPU hanya membutuhkan waktu selama 12,04 milisekon. Nilai tersebut sama dengan waktu yang didapat dari pengujian pada GPU *server*.

Pengujian kemudian dilakukan pada mesin virtual kedua, yang diberi nama **ubuntu2**. Metode yang digunakan sama persis dengan pengujian pada mesin virtual pertama (**ubuntu1**). Perbedaan dari kedua mesin virtual tersebut terletak pada perangkat keras dari *compute node* tempat mesin virtual berjalan. **Ubuntu2** berjalan pada *compute node 2*, dimana *compute node* tersebut memiliki spesifikasi CPU yang lebih tinggi serta lebih baru dari *compute node* pertama, sehingga dapat menimbulkan perbedaan kinerja. Data pengujian pada mesin virtual **ubuntu2** dapat dilihat pada tabel 4.9.

Dari data pada tabel diatas, mesin virtual **ubuntu2** membutuhkan waktu sekitar 12 detik (11934 milisekon) dengan menggunakan satu CPU *core*. Dengan menambahkan dua buah CPU *core*, waktu yang dibutuhkan berkurang sebesar 50 persen menjadi 6 detik (6132 milisekon). Dengan menggunakan empat buah CPU *core*, waktu yang dibutuhkan kembali berkurang sebesar 50 persen menjadi 3 detik (3033 milisekon). Peningkatan tersebut sesuai dengan

Tabel 4.8: Spesifikasi Mesin Virtual **Ubuntu1** pada Pengujian Komputasi

Spesifikasi	Keterangan
Jumlah Prosesor	6
Kecepatan Prosesor	1.86 GHz
Metode Komputasi GPU	CUDA secara virtualisasi menggunakan RCUDA

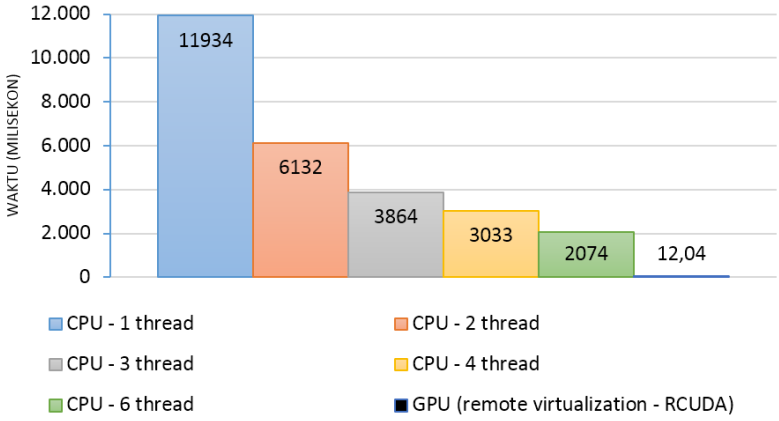
Tabel 4.9: Waktu Kalkulasi Perkalian Matriks pada Mesin Virtual **Ubuntu2**

Jumlah CPU *Core*	Waktu (milisekon)
1 *Core*	11934 ms
2 *Core*	6132 ms
3 *Core*	3864 ms
4 *Core*	3033 ms
6 *Core*	2074 ms
GPU (CUDA)	12,04 ms

peningkatan yang terjadi pada pengujian performa komputasi pada GPU *server* serta mesin virtual pertama (**ubuntu1**).

Khusus pada mesin virtual **ubuntu2**, ditambahkan pengujian menggunakan enam buah CPU *core* untuk dibandingkan dengan nilai-nilai sebelumnya. Dengan menggunakan enam buah CPU *core*, perkalian matriks membutuhkan waktu selama 2 detik (2074 milisekon). Nilai tersebut hanya 30 persen lebih cepat dari penggunaan empat buah CPU *core*, dan hanya 46 persen lebih cepat dari penggunaan tiga buah CPU *core*. Hal ini menunjukkan adanya *diminishing returns* pada penggunaan CPU *core* sebanyak lebih dari empat buah.

Pada pengujian komputasi perkalian matriks menggunakan GPU, didapat waktu yang dibutuhkan sebesar 12,04 milisekon. Nilai ini sama dengan waktu yang diambil pada GPU *server* serta mesin virtual pertama (**ubuntu1**). Hal ini menunjukkan bahwa proses



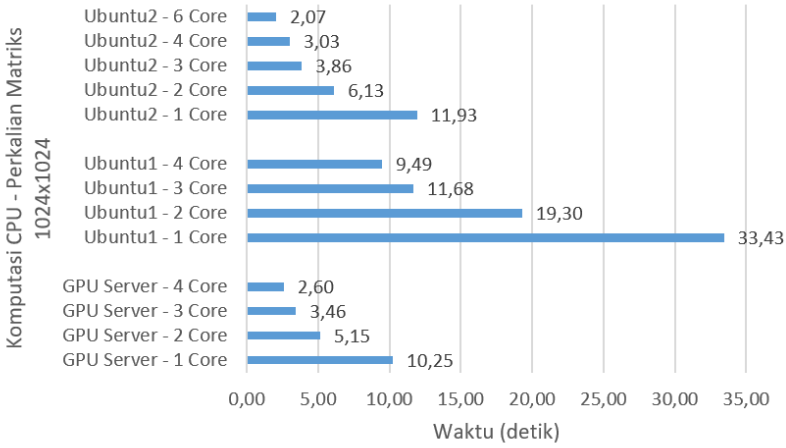
Gambar 4.30: Grafik perbedaan waktu pemrosesan perkalian matriks pada *Ubuntu2*

komputasi yang menggunakan GPU, baik secara langsung maupun melalui virtualisasi, akan bersifat independen dari perangkat keras yang digunakan pada sistem, selain dari GPU yang digunakan.

Dari rekapitulasi hasil pengujian komputasi CPU pada gambar 4.31, tampak bahwa mesin virtual **ubuntu1** membutuhkan waktu yang lebih lama untuk menyelesaikan perkalian matriks dibandingkan dengan GPU *server* dan mesin virtual **ubuntu2**.

Salah satu penyebabnya yaitu perangkat keras yang digunakan oleh **ubuntu1** pada *compute node 1* memiliki umur yang lebih tua dari perangkat keras yang digunakan oleh GPU server dan mesin virtual **ubuntu2** pada *compute node 2*, dimana perangkat keras yang lebih baru memiliki optimasi pada kemampuan pengerjaan instruksi per CPU *clock* (*Instruction per Clock*, IPC). Sehingga, dengan menggunakan kecepatan CPU yang sama, CPU terbaru akan bekerja lebih cepat bila dibandingkan dengan CPU generasi sebelumnya.

Dikarenakan pengujian GPU memberikan besar nilai waktu yang sama, maka dilakukan pengujian tambahan dengan memperbesar ukuran perkalian matriks menjadi 4096 x 4096. Pengujian hanya dilakukan pada GPU *server* dan salah satu mesin virtual.



Gambar 4.31: Grafik Rekapitulasi hasil pengujian komputasi CPU

Selain itu, pengamatan tambahan dilakukan pada waktu komputasi perkalian pada GPU dan waktu eksekusi program secara keseluruhan.

Tabel 4.10: Data Waktu Komputasi Perkalian Matriks 4096x4096 pada GPU dan Waktu Eksekusi Program Komputasi Keseluruhan

Letak Pengujian	**Waktu Komputasi pada GPU**	**Waktu Eksekusi Program**
GPU *server*	756.209ms	3 menit 48 detik
Mesin Virtual	756.214ms	3 menit 50 detik

Dari data pengujian tambahan yang telah didapatkan seperti pada tabel 4.10, bahwa dari waktu komputasi yang dijalankan pada GPU, tampak tidak ada perbedaan waktu yang berarti, yaitu sekitar 756 milisekon. Hal ini memastikan bahwa dengan penggunakan GPU secara konvensional maupun secara virtualisasi tidak memberikan perbedaan kinerja yang berarti.

Namun demikian, apabila diperhatikan dari waktu keseluruh-

an eksekusi program komputasi yang dijalankan, maka terlihat adanya perbedaan waktu eksekusi sebesar dua detik lebih lambat pada mesin virtual, apabila dibandingkan dengan eksekusi program pada GPU *server*.

Dengan mengutip kembali cara kerja dari RCUDA, maka data dan instruksi program dari memori mesin virtual dikirim terlebih dahulu pada GPU *server* untuk dikerjakan oleh GPU, kemudian dikembalikan lagi pada memori mesin virtual melalui jaringan setelah komputasi selesai dikerjakan. Dari pengiriman data antara mesin virtual dengan GPU *server* tersebut, terdapat penyempitan atau penurunan besar *bandwidth* pada jalur data antara memori dengan GPU (dalam hal ini mesin virtual dengan GPU *server*), karena data harus melewati jaringan *ethernet* agar sampai pada GPU *server*, sehingga dapat berpotensi menyebabkan terjadinya *delay* pada pemindahan data dari memori sistem pada mesin virtual menuju GPU *server* dan sebaliknya. Untuk memeriksa penurunan *bandwidth* tersebut, akan dijelaskan pada pengujian `bandwidthTest`.

Pengujian terakhir yang dilakukan adalah menjalankan program `bandwidthTest` untuk mendapatkan besar *bandwidth* antara memori sistem pada mesin virtual dengan memori GPU. Karena mesin virtual mengakses GPU secara virtualisasi melalui jaringan *ethernet*, maka dapat dipastikan akan terjadi /textitbottleneck pada jaringan yang menyebabkan kecepatan perpindahan data dari memori pada mesin virtual menuju memori GPU dan sebaliknya akan terhambat. Hasil pengukuran ditulis pada tabel 4.5. Tampak bah-

Tabel 4.11: Hasil pengukuran `bandwidthTest` pada Mesin Virtual

Jenis Transfer	***Bandwidth* (MB/s)**
Host to Device	115 MB/s
Device to Host	117 MB/s
Device to Device	108814 MB/s

wa besar *bandwidth* antara memori sistem (*Host memory*) dengan memori GPU (*Device Memory*) mengalami penurunan yang signifikan dibandingkan dengan nilai *bandwidth* yang didapatkan dari

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla S2050
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     113.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     117.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     108814.6

Result = PASS
```

Gambar 4.32: Output dari program `bandwidthTest` pada Mesin Virtual

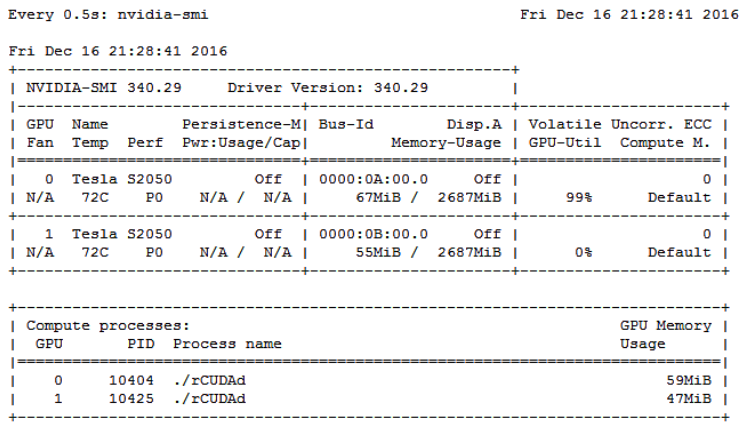
pengukuran pada GPU *server* pada tabel 4.2. Pada GPU *server*, besar *bandwidth* transfer data antara memori sistem dengan memori GPU mencapai kisaran 3 GB/s (gigabyte per detik), sementara pada mesin virtual, besar *bandwidth* hanya dapat mencapai maksimum 117 MB/s (megabyte per detik).

Kemungkinan penyebab utama dari penurunan *bandwidth* transfer data dari mesin virtual menuju memori GPU dan sebaliknya adalah kecepatan maksimum dari jaringan *ethernet*, yang secara teoritis hanya dapat mencapai 1 Gigabit per detik (sekitar 125 Megabyte per detik), sehingga menyebabkan terjadinya *bottleneck* pada jalur komunikasi antara sistem dengan GPU.

Hal ini dapat berpotensi mengurangi performa dari virtualisasi pada GPU apabila komputasi dilakukan menggunakan CPU dan GPU secara bersamaan, karena CPU harus menunggu data yang telah diolah GPU untuk selesai dikirim menuju memori sistem, dan sebaliknya. Namun, hal ini tidak berpengaruh apabila komputasi dilakukan dengan hanya menggunakan GPU ataupun CPU saja, seperti pada pengujian sebelumnya.

4.4.3 Pengamatan terhadap Penggunaan GPU

Selain beberapa pengujian komputasi, dilakukan pula pengamatan terhadap penggunaan GPU pada saat digunakan oleh mesin virtual sebagai GPU virtual. Pada skenario pengerjaan perkalian

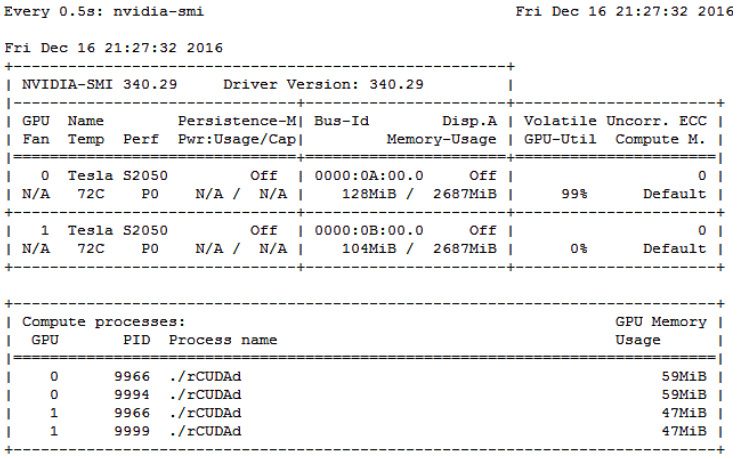


Gambar 4.33: Pengamatan penggunaan GPU saat digunakan oleh satu mesin virtual

matriks oleh satu mesin virtual, tingkat utilisasi salah satu GPU pada program `nvidia-smi` mencapai 99 persen seperti pada gambar 4.33. Hal ini menunjukkan bahwa GPU tersebut bekerja pada beban penuh saat digunakan oleh satu mesin virtual untuk mengeksekusi sebuah program CUDA. Selain itu, dapat diamati pula bahwa program pada mesin virtual hanya menggunakan sebuah GPU dari dua buah GPU yang tersedia, menunjukkan bahwa program tidak dioptimalkan untuk menggunakan lebih dari satu GPU.

Skenario pengamatan selanjutnya adalah melihat penggunaan atau utilisasi GPU pada saat menjalankan komputasi perkalian matriks dari dua mesin virtual secara bersamaan. Pada skenario ini, utilisasi GPU 0 yang terbaca pada `nvidia-smi` tetap berada pada kisaran 99 persen, seperti pada saat digunakan oleh satu mesin virtual, seperti pada gambar 4.35.

Pada saat pengamatan, kedua mesin virtual mengalami penurunan performa komputasi, dimana pada saat keduanya menggunakan GPU secara bersamaan, waktu yang dibutuhkan untuk menyelesaikan eksekusi program `matrixMul` bertambah dari 12,04 milisekon menjadi sekitar 23,3 hingga 24 milisekon, seperti pada gambar 4.34. Hal ini menunjukan bahwa ketika GPU digunakan



Gambar 4.34: Pengamatan penggunaan GPU saat digunakan oleh dua mesin virtual

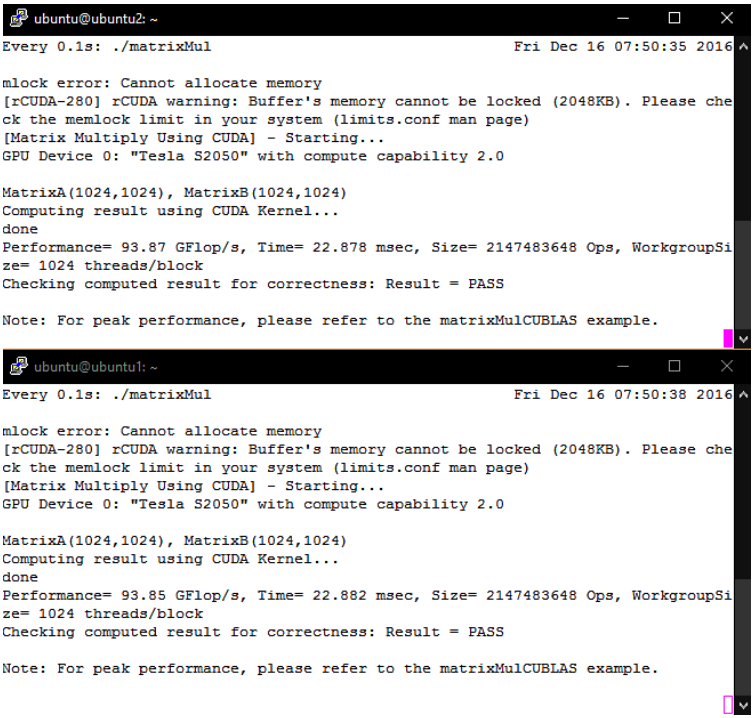
oleh dua mesin virtual secara bersamaan, maka kemampuan komputasi dari GPU tersebut berkurang menjadi separuhnya, sehingga waktu yang dibutuhkan oleh program pada mesin virtual bertambah menjadi dua kali lipat.

Penanggulangan dari masalah tersebut dapat dilakukan dengan mengalokasikan satu dari dua GPU pada NVIDIA Tesla terhadap masing-masing mesin virtual. Pengaturan dilakukan dengan mengubah *Environment Variables* berisi jumlah GPU virtual yang dapat digunakan oleh mesin virtual pada jaringan, serta alamat IP terhadap satu dari dua GPU yang tersedia pada GPU *server* seperti yang tertulis pada kode 4.6 dan 4.7. Tujuannya agar setiap mesin virtual hanya dapat mendeteksi dan menggunakan satu dari dua GPU yang tedapat pada jaringan.

```
RCUDA_DEVICE_COUNT=1      #Jumlah GPU Virtual yang digunakan
RCUDA_DEVICE_0=10.122.1.250   #Alamat IP GPU virtua
```

Kode 4.6: *Environment Variables* milik RCUDA pada **ubuntu1**

```
RCUDA_DEVICE_COUNT=1      #Jumlah GPU Virtual yang digunakan
```

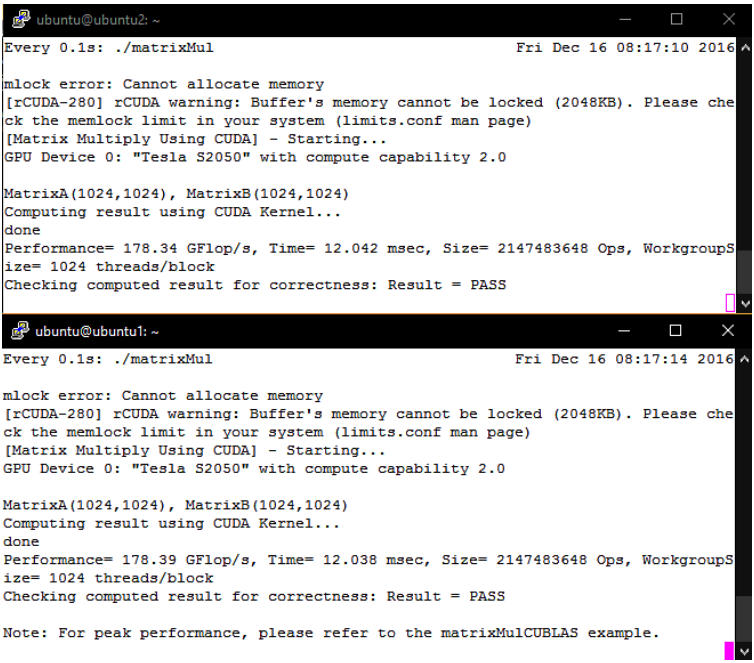


Gambar 4.35: Pengamatan terhadap waktu eksekusi program pada dua mesin virtual

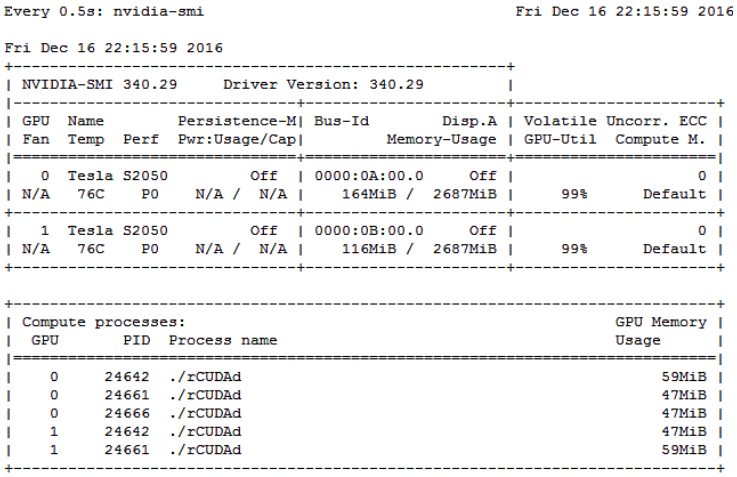
```
RCUDA_DEVICE_1=10.122.1.250:1 #Alamat IP GPU virtual
```

Kode 4.7: *Environment Variables* milik RCUDA pada **ubuntu2**

Dengan mengalokasikan satu GPU terhadap satu mesin virtual, maka penurunan performa dapat ditanggulangi. Hal ini ditunjukkan dengan waktu eksekusi program `matrixMul` yang kembali turun menjadi selama 12,04 milisekon, seperti pada gambar 4.36. Selain itu, penggunaan dari NVIDIA Tesla pada GPU *server* menjadi optimal, karena kedua GPU digunakan secara maksimal, seperti yang ditunjukkan pada gambar 4.37.



Gambar 4.36: Pengamatan terhadap waktu eksekusi program pada dua mesin virtual setelah optimalisasi



Gambar 4.37: Pengamatan penggunaan GPU saat digunakan oleh dua mesin virtual setelah optimalisasi

BAB 5
PENUTUP

5.1 Kesimpulan

Dari hasil yang didapat dari implementasi dan pengujian komputasi berbasis CPU serta virtualisasi GPU pada mesin virtual yang sudah dilakukan, maka dapat ditarik beberapa kesimpulan sebagai berikut :

1. Pada komputasi berbasis CPU, penambahan CPU *core* sebanyak dua kali lipat dapat mempercepat waktu komputasi sebesar 50 persen atau separuh dari waktu yang dibutuhkan. Hal ini menunjukkan bahwa CPU dengan jumlah *core* lebih banyak akan menyelesaikan proses komputasi dengan lebih cepat.
2. Mikroarsitektur dari CPU dapat memengaruhi kinerja komputasi dari CPU secara signifikan. Dengan kecepatan *clock* serta jumlah CPU *core* yang sama, ketiga jenis CPU yang digunakan pada pengujian memberikan performa komputasi yang berbeda, dimana CPU dengan mikroarsitektur generasi lama memiliki kemampuan komputasi yang lebih lambat daripada CPU berbasis mikroarsitektur terbaru.
3. Dengan menggunakan metode virtualisasi GPU jarak jauh, mesin virtual mendapatkan akses terhadap GPU untuk digunakan dalam melakukan komputasi GPU tanpa adanya penurunan performa yang signifikan apabila dibandingkan dengan penggunaan GPU secara konvensional. Namun demikian, masih terdapat adanya potensi penurunan performa komputasi, apabila komputasi yang dilakukan sangat bergantung pada kerjasama serta komunikasi antara CPU dengan GPU. Hal ini dikarenakan besar dari *bandwidth* antara mesin virtual dengan GPU dibatasi oleh *bandwidth* teoritis maksimum dari jaringan berbasis *ethernet*.
4. Pada virtualisasi GPU, pada saat GPU digunakan oleh lebih dari satu mesin virtual secara bersamaan, maka performa komputasi pada kedua mesin virtual tersebut akan mengalami

penurunan. Besar dari penurunan performa berbanding terbalik dengan jumlah mesin virtual yang sedang menggunakan GPU tersebut pada saat bersamaan.

5. Pada jaringan virtualisasi GPU dengan lebih dari satu GPU, masing-masing mesin virtual dapat dialokasikan untuk hanya menggunakan satu buah GPU pada proses komputasi untuk menghindari terjadinya penurunan performa akibat penggunaan satu GPU secara bersamaan.

5.2 Saran

Demi pengembangan lebih lanjut mengenai tugas akhir ini, terdapat beberapa saran yang dapat diberikan sebagai berikut :

1. Penggunaan perangkat keras, baik CPU serta GPU yang lebih baik, serta penambahan lebih dari satu GPU *server* dengan variasi GPU yang digunakan untuk melihat perbedaan kinerja komputasi pada GPU dengan mikroarsitektur berbeda.
2. Penambahan variasi jenis komputasi pada pengujian, serta penggunaan aplikasi *real-world* untuk melihat performa komputasi yang aplikatif, seperti simulasi dinamika molekuler (GROMACS atau LAMMPS).
3. Pengujian metode virtualisasi GPU dengan menggunakan *library* berbasis OpenCL.

BAB 6
Lampiran

```
b201@tesla-gw:~/NVIDIA_CUDA-6.5_Samples/bin/x86_64/linux/
    release$ ./deviceQuery
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static
    linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla S2050"
CUDA Driver Version / Runtime Version          6.5 / 6.5
CUDA Capability Major/Minor version number:    2.0
Total amount of global memory:                 2687 MBytes
    (2817982464 bytes)
(14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
GPU Clock rate:                                1147 MHz (1.15
     GHz)
Memory Clock rate:                             1546 Mhz
Memory Bus Width:                              384-bit
L2 Cache Size:                                 786432 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
    =(65536, 65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384),
    2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384,
    16384), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 32768
Warp size:                                     32
Maximum number of threads per multiprocessor:  1536
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024,
    64)
Max dimension size of a grid size    (x,y,z): (65535, 65535,
    65535)
Maximum memory pitch:                          2147483647
    bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2
    copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
```

```
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Enabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Bus ID / PCI location ID:           10 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice()
    with device simultaneously) >

Device 1: "Tesla S2050"
CUDA Driver Version / Runtime Version          6.5 / 6.5
CUDA Capability Major/Minor version number:    2.0
Total amount of global memory:                 2687 MBytes
    (2817982464 bytes)
(14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
GPU Clock rate:                                1147 MHz (1.15
     GHz)
Memory Clock rate:                             1546 Mhz
Memory Bus Width:                              384-bit
L2 Cache Size:                                 786432 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
    =(65536, 65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384),
    2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384,
    16384), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 32768
Warp size:                                     32
Maximum number of threads per multiprocessor:  1536
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024,
    64)
Max dimension size of a grid size    (x,y,z): (65535, 65535,
    65535)
Maximum memory pitch:                          2147483647
    bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2
    copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Enabled
Device supports Unified Addressing (UVA):      Yes
```

```
Device PCI Bus ID / PCI location ID:           11 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice()
    with device simultaneously) >
> Peer access from Tesla S2050 (GPU0) -> Tesla S2050 (GPU1) :
     Yes
> Peer access from Tesla S2050 (GPU1) -> Tesla S2050 (GPU0) :
     Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5,
     CUDA Runtime Version = 6.5, NumDevs = 2, Device0 =
    Tesla S2050, Device1 = Tesla S2050
Result = PASS
```

Kode 6.1: Output program deviceQuery

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>


#define SIZE 1024

int num_thrd;   // jumlah thread

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

// inisialisasi matrix
void init_matrix(int m[SIZE][SIZE])
{
int i, j, val = 0;
for (i = 0; i < SIZE; i++)
for (j = 0; j < SIZE; j++)
m[i][j] = rand() % 100;
}

// multithread function
void* multiply(void* slice)
{
int s = (long)slice;
int from = (s * SIZE)/num_thrd;
int to = ((s+1) * SIZE)/num_thrd;
int i,j,k;

printf("computing slice %d (from row %d to %d)\n", s, from,
    to-1);
```

```
for (i = from; i < to; i++)
{
for (j = 0; j < SIZE; j++)
{
C[i][j] = 0;
for ( k = 0; k < SIZE; k++)
C[i][j] += A[i][k]*B[k][j];
}
}
printf("finished slice %d\n", s);
return 0;
}

int main(int argc, char* argv[])
{
struct timeval  tv1, tv2;
gettimeofday(&tv1, NULL);

pthread_t* thread;  // pointer terhadap sekumpulan thread
int i;

if (argc!=2)
{
printf("Usage: %s number_of_threads\n",argv[0]);
exit(-1);
}

num_thrd = atoi(argv[1]);
init_matrix(A);
init_matrix(B);
thread = (pthread_t*) malloc(num_thrd*sizeof(pthread_t));

for (i = 1; i < num_thrd; i++)
{
// creates each thread working on its own slice of i
if (pthread_create (&thread[i], NULL, multiply, (void*)i) !=
    0 )
{
perror("Can't create thread");
free(thread);
exit(-1);
}
}

multiply(0);

// main thead
```

```
for (i = 1; i < num_thrd; i++)
pthread_join (thread[i], NULL);

free(thread);

// timer
gettimeofday(&tv2, NULL);
printf ("Total time = %f seconds\n",
(double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
(double) (tv2.tv_sec - tv1.tv_sec));

return 0;

}
```

Kode 6.2: Kode Program `matrix.c`

Halaman ini sengaja dikosongkan

DAFTAR PUSTAKA

[1] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí, "Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution," in High Performance Computing (HiPC), 2012 19th International Conference on, pp. 1–10, Dec 2012. (Dikutip pada halaman 1).

[2] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," CoRR, vol. abs/0910.1979, 2009. (Dikutip pada halaman 1).

[3] F. Silla, J. Prades, S. Iserte, and C. Reano, "Remote gpu virtualization: Is it useful?," in 2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), pp. 41–48, March 2016. (Dikutip pada halaman 2, 10).

[4] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," in Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, (New York, NY, USA), pp. 341–352, ACM, 2012. (Dikutip pada halaman 4).

[5] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," IEEE Transactions on Computers, vol. 61, pp. 804–816, June 2012. (Dikutip pada halaman 4).

[6] J. Duato, A. J. Pea, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in High Performance Computing and Simulation (HPCS), 2010 International Conference on, pp. 224–231, June 2010. (Dikutip pada halaman 4, 10).

[7] M. Dowty and J. Sugerman, "Gpu virtualization on vmware's hosted i/o architecture," SIGOPS Oper. Syst. Rev., vol. 43, pp. 73–82, July 2009. (Dikutip pada halaman 4).

[8] N. R. Widiyanto, Pengukuran Performansi Mesin Virtual pada Komputasi Awan. Institut Teknologi Sepuluh Nopember, 2015. (Dikutip pada halaman 4).

[9] S. Upstill, The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, 1990. (Dikutip pada halaman 6).

[10] "3d polygon rendering pipeline." http://www.cs.virginia.edu/~gfx/Courses/2012/IntroGraphics/lectures/13-Pipeline.pdf. Terakhir diakses pada tanggal 17 Januari 2017. (Dikutip pada halaman 7).

[11] "Common-shader core (windows)." https://msdn.microsoft.com/en-us/library/bb509580(VS.85).aspx. Terakhir diakses pada tanggal 9 Januari 2017. (Dikutip pada halaman 7).

[12] J. O. David Tarditi, Sidd Puri, "Accelerator: Using data parallelism to program gpus for general-purpose uses," tech. rep., October 2006. (Dikutip pada halaman 8).

[13] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From {CUDA} to opencl: Towards a performance-portable solution for multi-platform {GPU} programming," Parallel Computing, vol. 38, no. 8, pp. 391 – 407, 2012. {APPLICATION} {ACCELERATORS} {IN} {HPC}. (Dikutip pada halaman 8).

[14] B. Hess, C. Kutzner, D. Van Der Spoel, and E. Lindahl, "Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation," Journal of chemical theory and computation, vol. 4, no. 3, pp. 435–447, 2008. (Dikutip pada halaman 8).

[15] N. Chentanez and M. Müller, "Real-time eulerian water simulation using a restricted tall cell grid," in ACM SIGGRAPH 2011 Papers, SIGGRAPH '11, (New York, NY, USA), pp. 82:1–82:10, ACM, 2011. (Dikutip pada halaman 8).

[16] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," arXiv preprint arXiv:1410.0759, 2014. (Dikutip pada halaman 8).

[17] Y. Liu, B. Schmidt, and D. L. Maskell, "Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," BMC research notes, vol. 3, no. 1, p. 1, 2010. (Dikutip pada halaman 8).

[18] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on, pp. 65–68, IEEE, 2007. (Dikutip pada halaman 8).

[19] "Parallel programming and computing platform — cuda — nvidia." `http://www.nvidia.com/object/cuda_home_new.html`. Terakhir diakses pada tanggal 15 Oktober 2016. (Dikutip pada halaman 8).

[20] "Openstack installation guide for ubuntu 14.04." `http://docs.openstack.org/juno/install-guide/install/apt/content/index.html`. Terakhir diakses pada tanggal 30 Oktober 2016. (Dikutip pada halaman 11).

Halaman ini sengaja dikosongkan

BIOGRAFI PENULIS

Rizki Bayu Baskoro, lahir pada 18 April 1994 di kota Surabaya, Jawa Timur. Penulis menyelesaikan studi di SMP Negeri 6 Surabaya pada tahun 2009, kemudian melanjutkan pendidikan ke SMA Negeri 1 Surabaya dan lulus pada tahun 2012. Penulis kemudian melanjutkan pendidikan Strata satu di Jurusan Teknik Elektro, Fakultas Teknologi Industri ITS Surabaya pada bidang studi Teknik Komputer dan Telematika. Selama masa perkuliahan, penulis aktif sebagai asisten laboratorium Telematika B201 serta pernah menjabat sebagai ketua *development group* di bidang *networking*.

Halaman ini sengaja dikosongkan